MediateSpace

Applying Contextual Mediation to the Tuple Space Paradigm

Danny Matthews
Foundations of Software Systems Group
School of Informatics
University of Sussex

A thesis submitted, on September 30, 2014, in partial fulfilment of the requirements for the degree of Doctor of Philosophy (PhD) in the School of Informatics of the University of Sussex.

For Angela and my family

Statement of Originality

This thesis is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and the Acknowledgments.

This thesis is not substantially the same as any that I have submitted or am currently submitting for a degree, diploma or any other qualification at any other university. No part of this dissertation has already been, or is being submitted for any such degree, diploma or qualification.

September 30, 2014

Abstract

I designed, implemented and evaluated a decentralised context-aware content distribution middleware. It can support a variety of applications, with all network communication handled transparently behind a tuple space based interface. Content is inserted into the network with an associated condition stipulating the context that must be matched to receive it. Conditions can be expressed using conjunctions, disjunctions, a form of universal and existential quantification and nested block scopes. Conditions are mapped onto a set of spatial indexes to enable lookup; and these are inserted into a distributed multi-dimensional spatial data structure (e.g. an R-Tree). They are also translated into an OWL representation to enable evaluation.

Nodes bind to their most geographically proximate neighbours which allows distance-sensitive context sharing. The middleware is capability-aware, pushing computationally expensive tasks onto more capable nodes.

I evaluated my system through benchmarks and simulation, defining condition classes which collectively represent a large portion of the condition space. Random conditions were generated from these classes. Node mobility was controlled through a number of probability distributions. Benchmark evaluation times were reasonable, evaluating 500 typical messages in 1.4 seconds each. When the number of stored contexts were reduced, this improved dramatically, evaluating 500 much more complicated conditions in one-tenth of a second each. The number and complexity of context parameters has a major impact on efficiency.

The number of spatial indexes generated was reasonable for most conditions, with a 95th percentile of 6. However, existential quantification was a challenge for both condition evaluation and index generation due to the potentially large number of possible combinations of conditions.

As expected, simulations found that the distribution of workload was very uneven because nodes tend to cluster in large cities; meaning that most communication is localised within these areas. Also, node density had a dramatic impact on the number of received messages as nodes within sparse areas were unable to obtain context information which precluded condition evaluation.

I achieved my research goals of developing a distributed context-aware content distribution framework.

Acknowledgments

I would like to thank my supervisors Dr Dan Chalmers and Dr Ian Wakeman for their invaluable support throughout my research; offering encouragement, direction and valuable feedback. I would also like to thank Dr Des Watson for his support and for always being happy to help.

I am also thankful to Dr Simon Fleming for his copious feedback on paper drafts, to Dr Renan Krishna for his advice and encouragement and to Dr Shinya Sato for helping me to get through the madness that was printing and binding.

Lastly, but certainly not least, I would like to express my thanks to Angela for her constant encouragement and to my family for the unfailing support they have shown me throughout my life.

Contents

List of Figures

List of Tables

Listings

1 Introduction

Weiser's vision of ubiquitous computing [90] is closer to being fulfilled than ever before, with great strides towards it having been achieved in recent years. It was estimated in 2009 that there were approximately four billion pervasive devices in circulation, with numbers growing rapidly [80]. These devices almost uniformly provide built-in sensors such as cameras, microphones, GPS and wireless communication capabilities such as 802.11, 4G and Bluetooth. People are becoming ever more comfortable with the use of sensors, and there is every reason to expect this familiarisation to grow as time goes by. This growth in availability and public understanding opens the way for a variety of new applications and services. Some have visions of harnessing these sensors on a city or even country wide scale to create new applications that will improve the lives and living conditions of all [80].

Applications may simply aim to make certain things more convenient or more enjoyable for users. One example is LocoMatrix[1] who provide location-based games for fun and education. Other works focus on more serious applications such as healthcare monitoring [57] and advertising [29, 47].

Our work aims to support the development of applications such as these through the development of a *distributed context-aware content distribution framework*. We provide an overview of our system in section 1.3.

We will now briefly discuss the main contributions of our work and provide an overview of our proposed framework. We then provide a motivation for our framework by proposing two possible real-world applications. Finally, we describe the organisation of the remainder of this thesis.

1.1 Contributions

- **A Context-Aware Language**
  For supporting the development of Context-Aware applications.

[1] http://www.locomatrix.com/

- **A Context-Aware Middleware**
  A distributed and scalable context-aware content distribution middleware.

- **Contextual Condition Spatial Indexing Algorithm**
  An algorithm for mapping our contextual condition language to a multi-dimensional spatial index such as the R-Tree [46].

- **Contextual Language OWL Representation**
  An OWL representation of our context-aware language

- **A Context-Aware Framework Taxonomy**
  A taxonomy for comparing frameworks along the dimensions of flexible evaluation, ontology extension, heterogeneous interoperability and decentralisation.

Each of the above contributions are the candidates' own work.

Our contributions are discussed briefly in Section 1.3. The context-aware language and middleware are discussed in detail in Chapter 4 and the contextual condition indexing and OWL representation algorithms are discussed in Chapters 5 and 6 respectively.

We would also like to acknowledge the work of others on which our work depends.

Tuple spaces were proposed by Gelernter [38] and enhanced with context-awareness by Murphy el al. [23, 25, 27].

R-Trees were proposed by Guttman [46] and refined by a number of researchers including Berchtold et al. who created the more efficient X-Tree [10] and Bianchi who designed and implemented a distributed form of R-Tree [11]. This was later extended with fault tolerance by Valero et al. [87].

Chalmers et al.'s work on context-aware mediation [15, 16] and the plethora of context-aware middleware solutions were also an influence on our work. In particular we wish to highlight the Context Toolkit [78] for helping to establish the field, the MetroSense project [32] for proposing the concepts of network symbiosis, asymmetric design and localised interaction, the Hydrogen framework [50] for incorporating simple decentralisation and the SOCAM [45] and TOTAM middlewares [13] for their incorporation of OWL and tuple spaces respectively.

The human mobility models of Barabasi et al. [6, 42], Wang et al. [88] and Newman [65] were instrumental in the design of our simulation models.

The Web Ontology Language (OWL) [53] was used to implement our context-aware language.

We also acknowledge the work of application researchers on whose software we used. These are the PlanetSim network simulator [3], the JMH benchmarking tool [68], the Gephi data visualisation software [8], the FaCT++

OWL reasoner [85] and the OWL API [51].

1.2 Published Works

MediateSpace: decentralised contextual mediation using tuple spaces, Proceedings of the Third International Workshop on Middleware for Pervasive Mobile and Embedded Computing, M-MPAC 2011 [59]

Improving the Effectiveness of Advertising Through Contextual Mediation,
The 5th Workshop on Pervasive Advertising, Pervasive 2012 [60]

1.3 System Overview

The MediateSpace system is a middleware application designed to provide decentralised context-aware content distribution. Shared data are referred to as Messages and are distributed across the nodes of the network. Messages are designed to be general, allowing data to be represented using an arbitrary structure. Each Message is associated with a contextual condition which stipulates the context that the user must match in order to receive it. Messages are indexed using these contextual conditions, allowing users to specify contextual queries which are used to lookup relevant messages in the network.

Users can also issue requests for context information that they cannot access locally via Context Request messages issued to geographically proximate nodes.

1.3.1 Contextual Conditions

Contextual conditions can be constructed using simple conjunctions and disjunctions, or by using a modified form of the $\forall$ and $\exists$ predicate statements. They also support block scoping and arbitrarily nested blocks.

1.3.2 Modelling Context

The *Context* tuple allows some aspect of context (e.g. location or temperature) to be defined abstractly and *ConcreteContext* tuples provide a concrete implementation of a Context. This allows systems to have a shared understanding of the semantics of a particular context without enforcing a particular type of sensor or implementation.

Context and *ConcreteContext* tuples are analogous to object-oriented interfaces and classes respectively. Each Context structure defines one or more *Contracts* which roughly equate to methods in an interface. An ontology may also be specified if relevant. ConcreteContext structures implement Contexts,

providing a concrete implementation for each of the Contracts and ontology. Within a ConcreteContext, Contracts are roughly equivalent to static class methods.

Contract parameters and return values support six data types: (`Boolean`, `String`, `Integer`, `Double`, `Date`, `Time`, `Ontology`)

1.3.3 Messages

Message tuples specify a contextual condition (discussed below) which must be satisfied by a requesting party if they wish to receive the message. The message payload is stored within any number of modules and corresponding Adverts can be defined for each module. These adverts are used to inform the requesting party as to the contents of the module; allowing them to decide whether they wish to receive it. Message evaluation is performed by translating the contextual condition within the message into an OWL representation. This representation is then evaluated using an OWL reasoner.

1.3.4 Network Nodes

MediateSpace differentiates between two types of node in the network: Regional and Participant. Regional nodes are responsible for any computationally expensive operations which need to be performed and are also responsible for the majority of network communication. Hence, it is intended that the more capable machines on the network are used to host Regional nodes (e.g. server machines). Participant nodes will usually represent mobile devices such as phones and tablets. They may insert messages into the network, issue message and context requests and satisfy context requests from other Participant nodes.

1.3.5 Network Structure

Each Participant node is bound to it's most geographically proximate Regional node and performs rebinds periodically to ensure that it is still bound to a nearby device. Regional nodes bind to the n most geographically proximate Regional nodes. This allows them to issue context requests to one another in an attempt to retrieve context information from an otherwise unreachable but nearby Participant node. An OWL representation of the Context and ConcreteContext structures is used by the Regional node to establish which of their neighbours can provide the desired type of context. This topology is summarised in Figure 1.1.

This geographical awareness is harnessed to ensure that any context information shared between nodes is valid, as most context will lose its relevance if used too far from its origin.

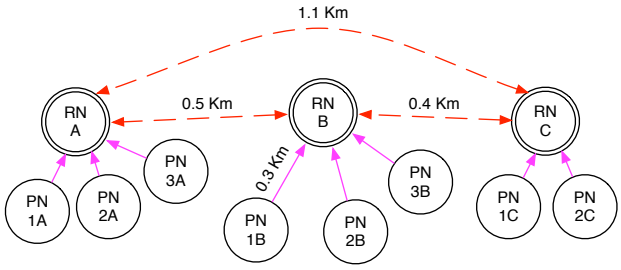


Figure 1.1: MediateSpace Topology

1.3.6 Node Communication

Each node has access to two tuple spaces: an internal space and an external space. Communication between nodes is conceptually handled by placing addressed tuples into these spaces. This dual tuple space abstraction makes communication straightforward for the end user but in reality communication relies on an overlay network and the distributed spatial indexes discussed in the next section.

1.3.7 Spatial Indexing

Messages and Regional node locations are stored in two separate distributed multi-dimensional spatial indexes. In this work we use an R-Tree [11, 46] but other indexes could be used such as an X-Tree [10] or TV-Tree [56].

We have devised an algorithm for mapping our contextual condition language to a multi-dimensional spatial index. We provide mappings for all aspects of the language including the use of conjunctions, disjunctions, blocks, nested blocks and existential quantification ($\forall$, $\exists$). This algorithm allows us to index our messages (or any other data) via a contextual condition and to efficiently lookup these messages within a spatial index using a contextual query.

1.4 Motivation

The generality of the Message format in our middleware allows it to be applied to a number of applications such as context-aware games or pervasive advertising. We will now provide a motivation for our system by discussing these two potential applications.

1.4.1 Pervasive Advertising

The Pervasive Advertising application discussed in this subsection is based on our 2012 paper on the same subject [60].

Existing pervasive advertising frameworks such as MyAds [29] and MobiAd [47] focus primarily on matching adverts based on explicitly entered information (such as user demographics and interests), derived information (e.g. through the parsing of browser history, Facebook or E-Mail) and location. A number of commercial mobile services also have this focus[2,3]. We aim to demonstrate how these types of context can be represented using the MediateSpace middleware.

Our example of use considers a shopping scenario where stores wish to advertise their products to appropriate customers. They wish to target individuals based on their budget, their proximity to the store, their availability and their shopping interests. Our system allows stores to distribute adverts with contextual conditions attached; delivering the adverts to only those individuals whose context matches. Figure 1.2 illustrates a potential design for this scenario with five Contexts and seven ConcreteContexts.

We discuss this example scenario in much more detail when discussing our context-aware middleware in Chapter 4.

1.4.2 Distributed Geocaching

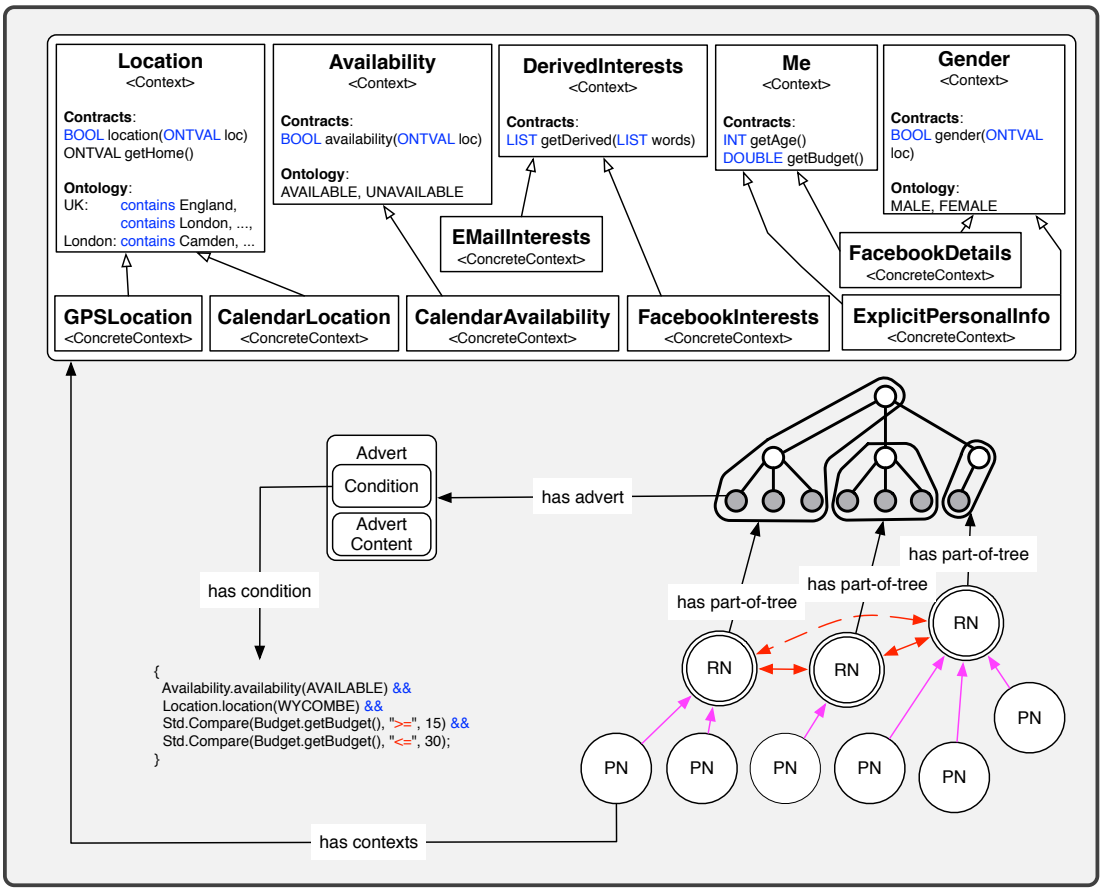
Geocaching[4] can be described as a modern form of treasure hunt, where people hide containers of varying size in the environment and register the GPS location of the container online. Players then use a GPS device to locate the container and acknowledge finding it by making a note in a log book within the container and online.

At present all geocaches are stored centrally at geocaching.com. Our system could be used to decentralise their operation.

1.4.2.1 Representing Geocaches

A geocache is represented as a set of GPS co-ordinates and a number of fields. These fields are the description, a list of recent log book messages, a hint to find the cache, a collection of photos and a set of attributes (discussed in the following section). The description and hint could be represented as simple strings, while the logs and photos could be represented as lists of strings and binary data respectively. In order to reduce the amount of data which needs to be transmitted at one time the log and photo data could contain only a subset of the available entries and the name of a tuple containing the next subset could be appended to the end of the data in case the user wishes to retrieve them. This method could be applied continuously to form a tuple based linked list of the data.

[2] http://www.admob.com/
[3] http://advertising.apple.com/uk/
[4] http://www.geocaching.com/



Context	Description
Location	Provides an ontology which represents different regions of the United Kingdom. **location(ONTVAL)**: Accepts an ontology value as parameter and returns a boolean indicating whether the user is within the specified location. **getHome()**: Returns the ontology value that represents the user's place of residence. **ConcreteContexts**: Using GPS or the users' calendar.
Availability	**availability(ONTVAL)**: Accepts an ontology value as parameter and returns a boolean indicating whether the user is currently available. **ConcreteContexts**: Via the users' calendar.
DerivedInterests	**getDerived(LIST)**: Accepts a list of words taken from a data source and outputs a list of derived words. **ConcreteContexts**: Data mining the users' E-Mail or Facebook account.
Me	**getAge()**: A single Contract for obtaining the age of the user. **getBudget()**: Returns a floating-point value representing the budget of the user. **ConcreteContexts**: Explicit information input by the user or via Facebook.
Gender	**gender(ONTVAL)**: Accepts an ontology value as parameter and returns a boolean indicating whether the user is of a specified gender. **ConcreteContexts**: Explicit information input by the user or via Facebook.

Figure 1.2: A Summary of our Pervasive Advertising Application

1.4.2.2 Attributes

In addition to a description and hint each cache can have a number of attributes associated with them. These attributes include (among many others) the availability of the cache (e.g. during the daytime, night, 24/7), the seasons of the year that the cache is available, the approximate amount of time it takes to find and the amount of walking required.

Geocaching.com divides walking time into three categories:

- Short Hike ($<$ 1 km)
- Medium Hike (1 - 10 km)
- Long Hike ($>$ 10 km)

This could be represented using an ontology with one concept for each category. However, we could improve the granularity of this attribute by representing distance as a floating-point value, allowing users to search for caches within any range.

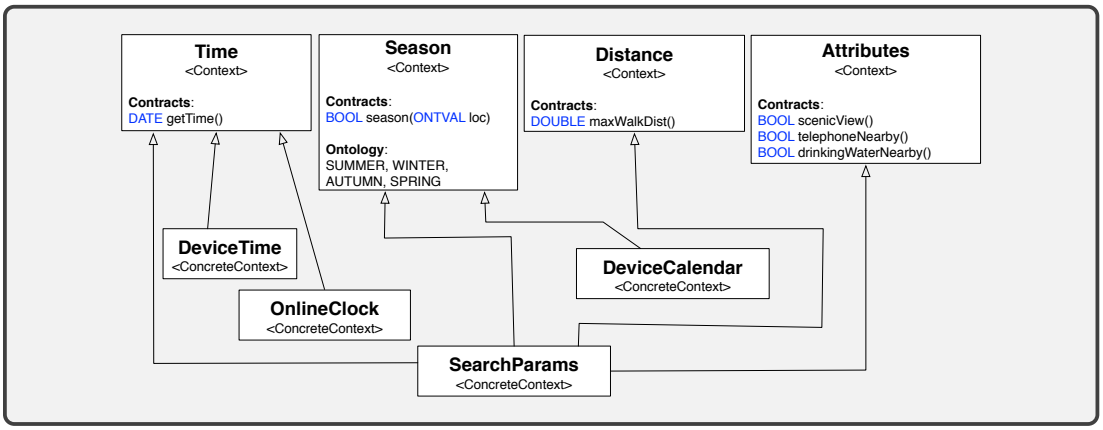
Some of the other attributes do not have a natural mapping to an ontology or numeric value and equate to a boolean (e.g. a cache has a "scenic view" or "telephone available" or it does not). These could be represented as individual Contracts which return boolean values or alternatively they could be used for further filtering after lookup in a similar way to our handling of keywords in the Pervasive Advertising example discussed later in this chapter.

1.4.2.3 An Example

We have defined a number of Contexts and ConcreteContexts that may be defined for a distributed geocaching application. These have been summarised in Figure 1.3. The SearchParams ConcreteContext is useful as it allows users' to specify their own values for a Contract. For example, if a user was attending a conference in one month's time and they wanted to do some geocaching during their stay they could specify the GPS co-ordinates of the conference centre and set the Time and Season appropriately to search for caches.

Figure 1.4 presents an example of how a geocache may be represented in our system. The specified condition stipulates the location of the geocache on the grounds of the University of Sussex and specifies that the following must be true in order for the cache to be returned when a query is issued:

1. The time must be between 8 AM and 5 PM
2. It cannot be winter
3. The maximum distance the user is willing to walk must be greater than or equal to 10 Km
4. The user must not have rejected a scenic view.



Context	Description
Time	Represented as a single Contract for obtaining the time. **ConcreteContexts**: Reading from the device clock, an online service or via SearchParams.
Season	Represented as a single Contract for obtaining the current season. **ConcreteContexts**: Reading from the device calendar or via SearchParams.
Distance	Represented as a single Contract for obtaining the maximum walk distance to a cache. **ConcreteContexts**: Via SearchParams. This Context does not strictly represent a facet of the users context; instead it allows them to specify these attributes in order to restrict the search results.
Attributes	Each attribute we wish to represent should have it's own Contract returning a Boolean. In this example we are representing the "has scenic view", "has phone nearby" and "has drinking water" attributes. **ConcreteContexts**: Via SearchParams. Again, this Context does not represent a facet of the users context; being used instead to restrict the search results.

Figure 1.3: A Summary of our Geocaching Application

```
MessageTuple {
  Meta {
    ((tupleName, "cache-b8fa8a79847db01aa328"), (msgUniqueId, 865),
     (originatorId, "P2"), (sourceId, "P2"), (destinationId, "P5"));
  }
  Condition {
    @Location.XY(51.06, 0.127, 0.5);@
    {
      Std.Compare(Time.getTime(), ">", 08:00:00) &&
      Std.Compare(Time.getTime(), "<", 17:00:00) &&
      {
        Season.Season(SUMMER) || Season.Season(AUTUMN) ||
        Season.Season(SPRING)
      } &&
      Std.Compare(Distance.MaxWalkDist(), ">=", 10) &&
      Std.Compare(Attributes.scenicView(), "==", true)
    }

    Advert {
      (Meta, ["Basic Info",
              "Title, desc, hint, difficulty, terrain, size"]),
      (Logs, ["Logs", "Log info for this cache"]),
      (Photos, ["Photos", "Photos of the cache"]);
    }

    PayLoadModules {
      Meta { ((title, "University of Sussex"),
              (description, "The cache is on the boundary walk"),
              (hint, "in a tree"),
              (difficulty, 2), (terrain, 2), (size, 2));
      }
      Logs { [ (( title, "Found it!"), (date, 20/08/2014),
                 (body, "Solved. quick trip to find it. TFTC")),
               ((nextLink, "cache-444b7c605dd7e22fdac6"));]
      }
      Photos { [ ((photo1, "d76357f331433b1eec89c35e82b"),
                   (nextLink, "cache-1ea90295d182746de156"));]
      }
    }
  }
}
```

Figure 1.4: An Example Geocache Message

1.5 Thesis Organisation

The remaining chapters of this thesis are organised as follows:

- Chapter 2 discusses a number of decentralised protocols which each provide different benefits; and we relate each to the goals of our middleware.
- Chapter 3 defines Context and discusses how it may be categorised and represented. We also discuss a number of centralised and decentralised context frameworks and evaluate them according to the four key criteria we defined for our middleware. This is summarised using a taxonomy at the end of the chapter.
- Chapters 4, 5 and 6 discuss the main contributions of our work. Specifically, we discuss our MediateSpace middleware and the spatial indexing and OWL representation algorithms discussed in section 1.1.
- Chapter 7 discusses our system design and experimental setup. This includes the definition of variables and a number of experimental models to improve the realism of the simulation.
- Chapter 8 provides the results of our MediateSpace simulations, OWL evaluation benchmarks and an evaluation of our spatial indexing algorithm.
- Chapter 9 offers a summary of the thesis and suggests future work.

2 Decentralised Protocols

2.1 Introduction

This chapter reviews research into decentralised communication protocols. That is, we discuss networking protocols which allow communication between computers (which are often mobile devices such as phones or tablets) without the need for a centralised system to arbitrate the communication.

Decentralisation is a desirable trait because it distributes the system functionality across the nodes of the network; avoiding a single point of failure and potentially increasing the scalability of the application. Our middleware is decentralised in nature and this chapter discusses many of the decentralised protocols available in relation to the goals of our middleware.

Traditional point-to-point synchronous protocols are coupled according to space, time and synchronisation. We now provide a brief explanation of these terms:

Time Coupling Refers to the fact that all interacting nodes are required to be present in the network at the same time if they wish to communicate with one another.

Space Coupling Implies that if two nodes wish to communicate with one another they can only do so via an explicit contact address (e.g. IP address).

Synchronisation Coupling Refers to protocols which require the requesting node to wait for a reply from the requestee before they can do any further processing.

The distributed protocols we discuss below relax some or all of these couplings. Specifically, we discuss *Publish/Subscribe networks*, *Distributed Tuple Spaces*, *Distributed Hash Tables* and *Distributed Graphs* (with a specific focus on trees used for spatial indexing).

2.2 Publish-Subscribe Networks

Publish-Subscribe networks are a messaging middleware decoupled in space, time and synchronisation [33]. Subscribers register their interest in one or more events with an event service, which then notifies the subscriber asynchronously whenever a publisher pushes an applicable event to the service.

Publish-Subscribe networks are decoupled in space because all communication between publishers and subscribers is handled via an event service; thus, the nodes do not need to be aware of each other in order to communicate. Time decoupling is achieved as the event service allows subscribers to issue a subscription event even if some or all publishers are offline. Also, provided the event service retains a buffer of events a subscriber can receive events issued by a publisher even if the subscriber was offline at the time the event was originally sent. Subscribers issue requests to the event service and are then free to continue until the event service notifies them of an appropriate event; thus, Publish-Subscribe networks also achieve synchronisation decoupling.

Publish-Subscribe networks can be centralised, where the event service is situated on one or more servers or they can be implemented in a distributed fashion. For instance, Shvartzshnaider, Ott and Levy [81] proposed a content-based Publish-Subscribe network written over a Distributed Hash Table (DHT).

Publish-Subscribe networks tend to come in three flavours: topic-based, content-based and type-based [33]. These will now each be discussed in turn.

2.2.1 Topic-Based

Topic-Based networks use string identifiers to represent subscription event types and each topic is viewed as having its own event service with a unique name and operations for publishing or subscribing to the topic. For example, you may subscribe to events from the BBC Sports service by issuing a subscription request to the topic "uk-bbc-sport". Topic-Based networks have the advantage of enforcing platform interoperability as the event space is divided by simple strings.

Networks may use either a flat or hierarchical addressing scheme. With flat addressing, publications to a topic will only be received by subscribers who have explicitly subscribed to the given topic, whereas with hierarchical addressing topics can be organised according to containment relationships, where publishing to a topic will also deliver the event to any topics further up in the hierarchy. For example, say the network has three topics ("uk-bbc", "uk-bbc-sport" and "uk-bbc-comedy") and the latter two are contained within the former; if a user is subscribed to "uk-bbc" they will receive all events dispatched to any of the three topics. If instead they were subscribed to "uk-bbc-comedy", they would only receive events dispatched to "uk-bbc-comedy".

2.2.2 Content-Based

In Content-Based networks subscription events specify details of the actual content of the event rather than being based on some "predefined external criterion" [33]. Subscriptions are often based on internal attributes of the event or some event meta data. For example, in a stock notification system some internal attributes of an event may be company name or price [33].

Subscriptions are specified using a subscription language which is often key-value based with relational operators (e.g. $<$, $>$, $\geq$) and logical operators (e.g. AND, OR). These subscriptions are often specified as strings which are parsed according to a grammar by the event engine. However, they may also be issued using templates which declare a type and any number of attributes to be matched (similar to the way tuple matching occurs in a tuple space based middleware; see Section 2.3). Finally, subscriptions may be given as executable code specified as a predicate object to be evaluated against potential events on the event service. One disadvantage of the executable approach is that it makes it very difficult to optimise the network communication and event processing tasks.

2.2.3 Type-Based

Typed-Based networks often use object-oriented principles to represent both subscriptions and event types. It's main benefits are that it provides type safety and sub-typing. Also, type-based networks may be used to represent content-based filtering if public member fields are used to represent the internal or meta attributes of an event.

2.2.4 Summary

Eugster et al. [33] recommend choosing Topic-Based networks if the primary property of subscriptions range over a limited set of discrete values because of it's efficiency. However, if more flexibility is needed then Content-Based networks can be highly expressive but with the cost of requiring more sophisticated protocols and a higher overhead. They also suggest that Topic-Based networks can be combined with Content-Based networks in cases where the primary property is discrete and limited but further disambiguation is needed.

A content-based publish subscribe system could be used to process our contextual conditions and provide message lookup. However, we felt that the transient nature of context did not lend itself well to a subscription based approach as the subscriptions would likely become invalid quickly. Short expiry times could be specified for each request but that would negate the main benefit of the protocol.

2.3 Tuple Spaces

A tuple space is a shared space accessible to many entities (first proposed by David Gelernter [38]). These entities may be separate processes or separate machines communicating with the space over the network. Tuple spaces store tuples - which are simply packages holding arbitrary data. Each tuple is stored in the space with a set of fields whose values must be matched in order to retrieve it. A tuple space has three basic operations which can be performed on it:

- **out** - Place a new tuple in the space.
- **rd** - Read a copy of a tuple from the space (if available).
- **in** - Read and remove a tuple from the space (if available).

The rd and in commands are blocking operations, meaning that they will not return until a matching tuple is found. There are also non-blocking variations of these commands (rdp and inp) and versions for retrieving multiple matching tuples at a time (rdg and ing) [27]. Figure 2.1 illustrates the three basic operations of a tuple space.

The tuple space has the benefit of being decoupled in space because all communication is directed towards the tuple space rather than communicating directly with other nodes. It is also temporally decoupled because once a tuple has been inserted into the space it becomes associated with the space rather than the originating node. Thus, the tuple will remain even if the originating node leaves. This holds for subscribers also (i.e. nodes which consume tuples from the space) as tuples can be inserted into the space before the subscribers join. The issue of synchronisation decoupling is more complicated. Although this holds on the producer-side the semantics of the tuple space dictate that a tuple can only be removed (when the in command is invoked) by a single node. Thus, the consumer and tuple space must work closely with one other to ensure that the tuple is received on the consumer and removed from the tuple space atomically.

We have chosen to use the tuple space paradigm for our middleware as it provides spatial and temporal decoupling, and the tuple space abstraction provides a simple interface for application developers to interact with.

We now discuss a number of distributed variants.

2.3.1 LIME

Although several research projects have focused on designing distributed tuple spaces, one of the most mature of these is LIME (Linda In a Mobile Environment) [27]. LIME is a Java implemented middleware application which achieves the illusion of shared memory through the creation of a virtual tuple space defined as the union of the tuple spaces of all nodes within wireless communication distance. Their protocol ensures that all updates, additions and removals are reflected in the global tuple spaces of all nodes.

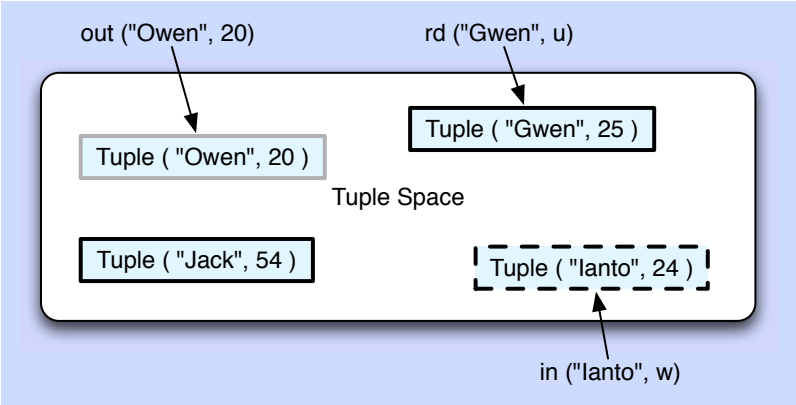


Figure 2.1: Illustrating the three basic operations of a tuple space.

They extended the semantics of the tuple space operations to reflect the distributed setting. For instance, the "out" operation may be augmented with a destination node and the "rd" and "in" operations may include both source and destination nodes.

LIME also supports the notion of "reactions". These are constructs which observe the properties of tuples as they enter the system and trigger the execution of an action when these properties match those specified by the reaction. Reactions may be specified as strong or weak. Strong reactions must execute their action immediately upon attaining a match whereas weak reactions do not have this constraint - the action may be executed at any time in the future.

LIME uses a distributed index called the Global Virtual Data Structure (GVDS) [4] which it uses to create and display a global tuple space containing the spaces of all nodes within communication distance.

2.3.2 TinyLIME and TeenyLIME

The TinyLIME middleware [25] is extended from LIME and focuses on the task of retrieving and aggregating sensor values from wireless motes. By using the distributed shared memory approach of LIME, the middleware facilitates the sharing of sensor values between nodes over the network. Sensor data may be retrieved as single values or as the aggregate of the sensor values over a specified time.

This approach differs significantly from the traditional method of collating sensor data (i.e. collecting all data into a central monitoring station), making it a more appropriate choice when sensors are sparse or isolated.

TinyLIME reactions have been enhanced to include contextual conditions. For example, it is possible to specify that a reaction should trigger only if a temperature value is between 20 and 30. TinyLIME is implemented using a

combination of the Java, C and nesC (Network Embedded Systems C) [37] languages.

TeenyLIME [23] provides very similar functionality to TinyLIME, with it's focus also being that of wireless sensor networks. However, this version of the middleware is designed to exhibit significant performance improvements as it is implemented on TinyOS[1] using the nesC and C languages exclusively.

2.3.3 Peer-To-Peer Tuple Spaces

A number of other peer-to-peer tuples spaces have been proposed. Like LIME, PeerWare [24] achieves distributed shared memory through the use of the Global Virtual Data Structure (GVDS) [4] which it uses to represent network nodes and documents (the payload). In order to maintain this shared state the GVDS data structures' of all available nodes are combined into one global space, with this information being shared via a peer-to-peer network. Primitives are provided to issue subscription requests to and to execute commands on subsets of nodes and documents. They note that the more complicated the resource language, the more complicated the runtime support architecture required on each node.

The peer-to-peer tuple space middleware [72] is designed to handle the task of resource brokering in a distributed fashion. All resource requests and resource usage information is shared via a tuple space which is spread over a Distributed Hash Table (DHT). The publish-subscribe pattern is used to match resource requests with appropriate resource nodes (e.g. resources having sufficient memory, being within the correct price range etc) and this co-ordination is handled via one or more service nodes.

Each Co-ordination service is structured using a three-tiered architecture:

Application Layer Allowing users to make requests and inform of updates.

Core Services Layer Responsible for matching tasks to subscribers and also for resource discovery (i.e. calculating the indexes for DHT lookups).

Connectivity Layer Delivers messages to the appropriate nodes using a DHT such as Chord.

Resources are identified by more than one attribute and so queries are N-dimensional. If a fixed value for each attribute is specified it is known as an N-dimensional point query whereas if ranges of values are specified then it is an N-dimensional window (or range) query.

The dimensions of a resource are both static (operating system, amount of memory etc) and dynamic (processor utilisation, physical memory utilisation, current price etc).

[1] http://www.tinyos.net/

2.4 Hash Tables

Associative arrays allow the mapping of keys to values. Using this data structure, you can subsequently search for a value by using the appropriate key as an index into the map. The simplest way to achieve this functionality is via a linear search where the search key is compared against each key in the array sequentially until the matching key is found (and the value returned). This is however slow if the size of the array is large with a worst case search complexity of O(n). An alternative approach is to represent the elements in a binary search tree. This improves on linear search with an average search complexity of O(log n) but again a worst complexity of O(n). The use of red-black trees [92] to ensure that the tree is balanced eliminates this worst case and gives a guaranteed O(log n) search complexity.

Although search trees have benefits such as support for ordered keys, if these benefits are unneeded you can achieve constant (O(1)) search complexity by using a hash table [43]. Hash tables operate by performing a calculation (called a hash function) on the key which produces a number between zero and the size of the table. The value is then stored at this index. Search then becomes a simple process of performing the same calculation on the search key and returning the value present at the corresponding index.

A perfect hash function will produce a unique index for every key inserted into the table. However, perfect hashing can be costly to achieve. A good hash function will distribute the keys evenly across the table but may cause collisions where more than one value is mapped to the same key. To guarantee a low number of collisions universal hashing can be used [12]. However, collisions are almost inevitable and in this case separate chaining can be used, where instead of each key mapping to a single value, it maps to a list of values. An alternative approach may be to perform a second hash on the key in the hope that this will produce a unique index [12]. If a very poor hash function is used search complexity can degrade to a worst case of O(n).

2.4.1 Distributed Hash Tables

Distributed Hash Tables allow the distribution of data over an overlay network, which achieves efficient lookup (O(log n) hops on average) with very little overhead as each node only needs to retain a small routing table of neighbours.

A number of overlays to achieve this task have been suggested. We summarise three such algorithms (Chord, Content Addressable Networking and Pastry) in the following sections.

2.4.1.1 Chord

In Chord [82], both nodes and keys hash themselves into an m-bit ID space using the SHA-1 algorithm which has good distributional properties. The

hash table is modelled as an identifier circle modulo 2^m, ensuring that all keys are mapped to the space regardless of size. When a node joins the network it takes over the management of X keys from its direct successor in the space and reassigns said keys when the node leaves. Key k is mapped to the node with the smallest ID larger than k. Each node maintains a routing table containing a maximum of m successors with power-two intervals around the ID circle. This scheme ensures that each node has significant local information while still being able to forward a query at least halfway along the remaining distance to the destination on each hop. The network is self-stabilising with each node periodically updating it's routing table.

2.4.1.2 Content Addressable Networking (CAN)

CAN [73] uses a d-dimensional cartesian co-ordinate space with each node claiming responsibility for a zone within this space. The co-ordinate space is represented on a d-torus which allows keys to wrap if their value exceeds the maximum dimensions of the space. Key-value pairs are deterministically mapped onto a point P using a uniform hash function and are stored at the node responsible for the region that P falls. Joining the network causes the joined node to split its region in half and allocate it to the new node. Each node maintains a neighbour list of those nodes in directly adjacent zones and greedy routing is used to forward a query progressively closer to the destination. The use of a d-dimensional cartesian space has the benefit of allowing the calculation of multiple paths to a destination which may be used when neighbours fail.

Routing path length can be reduced by increasing the number of dimensions because this increases the number of possible neighbours, and also improves fault tolerance. Another suggested extension are "realities", where the system maintains R independent co-ordinate spaces with each node holding R zones and R independent neighbour sets. Availability can be improved by storing the data on all realities and the number of hops can be reduced because each node now has r neighbour sets, making the discovery of nodes close to the destination more likely. It is possible to make the algorithm more topologically-aware and thus reduce latency by employing landmark routing techniques to group the regions of nearby nodes together.

2.4.1.3 Pastry

The overlay used by Pastry [76] is similar to that used by the Chord protocol, with each node being assigned an 128-bit ID using an algorithm which ensures a uniform distribution of nodes over the space (such as SHA-1). The space is also conceptually circular as in Chord. Messages are inserted into the node with the ID closest to the hashed value of the message. There are two main differences between Pastry and Chord. Firstly, Pastry takes into account the

geographical proximity of nodes, favouring closer nodes in the construction of the routing table. This helps to ensure an efficient route to the destination.

The second major difference is that while Chord regards an ID as indivisible, Pastry separates each ID into "a sequence of digits with base b^2". Routing tables are divided into levels with level n representing nodes that share the first n digits of the local node. The routing algorithm operates by first determining whether the message or query can be delivered in one step; if it can it is delivered. If it cannot be delivered in one step, the routing table is used to forward the message to a node whose ID shares a common prefix with the key by at least one additional digit. On each hop the length of the prefix grows and the number of matching nodes in the routing table decreases exponentially, which results in much larger distances being traversed on each additional hop. This allows the algorithm to deliver messages in O(log n) hops. Thus, Pastry has very similar lookup and query times to the other discussed DHTs' but also has the benefit of taking network locality into account which can lead to lower overall latency.

2.4.2 Applications

Hash tables are useful in any circumstances where you wish to be able to efficiently map a piece of information you own (the key) against a value that you need. Hash tables are enormously useful when programming an application and distributed hash tables have found uses in a wide number of situations, the most successful of which is probably peer-to-peer file sharing. It is used as an integral part of the popular BitTorrent [84] protocol for finding peers with access to a desired file in a decentralised way. BitTorrent is also used in industry to provide an inexpensive way for users to download software updates; one example being Blizzard Entertainment[2] who use it to distribute updates to the phenomenally successful World of Warcraft.

However, hash tables, distributed or otherwise have their limitations. Because they operate over a one-dimensional space it becomes extremely difficult to use them for any scenario requiring multiple dimensions. Although space-filling curves [77] can be used to map multiple dimensions to a single dimension they have been shown to quickly degrade in quality ([14], pg. 144). Therefore, in scenarios such as these it is advisable to use an alternative spatial data structure such as the R-Tree or one of its many variants.

2.4.3 Summary

As discussed, hash tables are an appropriate choice when data can be associated with a one-dimensional index. They may also be used with indexes of

[2] http://eu.blizzard.com/en-gb/

more than one dimension but this is often inadvisable as the quality of the mapping tends to degrade quickly.

Distributed hash tables are often a good choice for providing store and lookup facilities over the network. For instance, DHTs' provide time decoupling as data is usually stored on a different node than the one that introduced the data into the network. Thus, the data will continue to be available when the source node leaves. Also, DHT protocols provide a mechanism to pass the management of data to other nodes when one node leaves the network. Thus, the data will continue to be available even after the node originally tasked with managing the data leaves. DHT's provide spatial decoupling because each node is only responsible for retaining the addresses of a small number of other nodes in the network. Whether DHT's require synchronisation coupling depends on the implementation. Dabek et al. [26] propose a common API for use by DHT's (and other key-based routing protocols) which allow a node to be decoupled in terms of synchronisation. They provide callback mechanisms to be invoked on the local application when relevant data arrives so that processing can continue in the meantime.

As already discussed, DHTs tend to operate poorly when dealing with multiple dimensions. The indexing of our contextual conditions require multiple dimensions, with the necessary number of dimensions being equal to the total number of Contracts supported by the application. Hence, we have chosen not to use a DHT for indexing and instead use spatial indexes; which we discuss in detail in the next section.

2.5 Spatial Indexes

Spatial indexes allow you to efficiently index and lookup multi-dimensional data and are mainly constructed as trees. After a definition of terms this section discusses some of the many variants of spatial index and their distributed counterparts.

2.5.1 Graphs

Graphs ([43], pgs. 288-292) are generally defined as the pair G = (V, E) where V is a set of vertices and E is a set of edges. Vertices are used to represent an entity such as a computer or a person. Edges are used to indicate relationships between vertices and are represented as a line or arc between the aforementioned vertices. For example, an edge between two computers might be used to represent a direct network connection between the machines.

Graphs can be directed or undirected. In an undirected graph all edges represent a symmetric relationship between the vertices. For example, in the computer network example above it would indicate that both computers have a connection to one another. In a directed graph the edges have arrows

indicating the direction of the relationship. In our network example this may indicate that computer A has a connection to computer B but not vice versa. Edges may be unidirectional or bidirectional. Graphs may also be mixed, containing both directed and undirected edges, and in this case are usually represented as G = (V, E, A).

If the edges of a graph are weighted then they are each associated with a value which indicates some useful characteristic. In our networking example, each edge might be associated with a number estimating the latency of communication between computers. This weighting can then be used to aid graph traversal; perhaps by routing a message between two computers using the route with minimum latency. If a graph is unweighted then all edges are seen as having equivalent value.

A connected graph is a graph where every vertex can reach every other vertex along one of the edges, either directly or via a path consisting of several nodes.

2.5.2 Trees

Trees ([43], pgs. 75, 292-293) are the name given to the subset of connected graphs without cycles. They can be further delineated as free trees or rooted trees. Free trees have no ordering constraint whereas rooted trees have a single vertex labelled as it's "root". Rooted trees have directed edges leading away from the root and represent a hierarchy. Convention dictates that when drawn the root node is situated at the top of the graph with the non-root nodes below. Trees are often used to store key-value sets, with the key being used for indexing within the tree and the value being stored (or pointed to) from the corresponding node.

We now define the terminology commonly used with trees. A node (P) with a directed edge to one or more other nodes below it (collectively known as C) is known as the *parent* of C and C are known as the *children* of P. A node with zero children is known as a leaf, whereas a node with one or more children refers to an intermediate node. A balanced tree is a tree where all leaf nodes appear on the same level.

In some cases the tree is inverted so that directed edges lead towards the root and the root node is drawn at the bottom of the graph with the non-root nodes above.

See Figure 2.2 for an example of a connected graph and tree.

2.5.3 Spatial Indexes

This section first offers an overview of the precursor to spatial indexing techniques: B-Trees. We then go on to discuss several spatial indexing techniques and their variants. Specifically, we discuss *R-Trees*, *TV-Trees* and *X-Trees*.

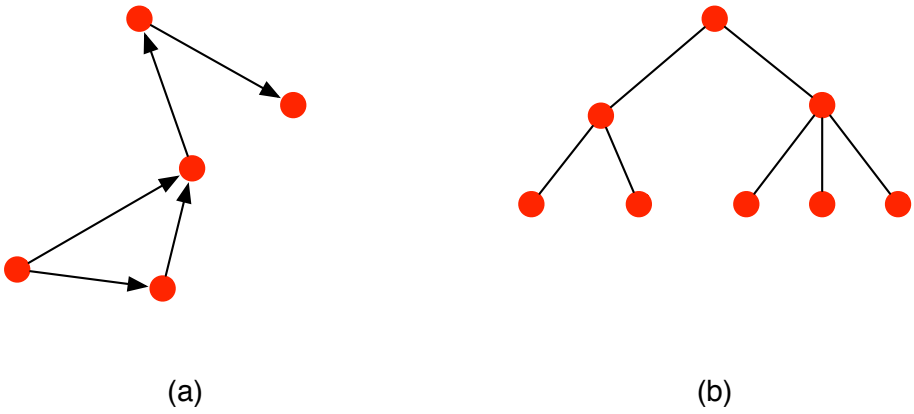


Figure 2.2: (a) A Directed Connected Graph (b) A Tree

2.5.3.1 B-Trees

The predecessor of spatial indexing trees is the B-Tree [22]. A B-Tree is a balanced tree where every node (except the root) has between M/2 and M children and the root node has between two and and M children if it is not a leaf. B-Trees are used to store any values that a total ordering can be defined for (e.g. numbers and strings). Each node holds a key which indicates its position within the tree relative to the keys of all other nodes.

The B-Tree is often used to provide indexing in DBMS software for several reasons:

1. The storing of many children (keys) per node allows them to be stored conveniently in blocks of memory so that fewer disk reads are necessary.
2. They allow efficient filtering operations to be performed such as "Give me all stored values > 25 and ≤ 50".
3. The use of partially full nodes makes insertion and deletion more efficient by reducing the amount of memory allocation and movement that needs to be performed.

For a node with k keys, it will have a maximum of k+1 child pointers which are used as separators for it's children. For instance, if a node contains the keys 7 and 16, it will contain three child pointers, with one pointing to a node containing keys less than 7, one pointing to a node greater than 7 but less than 16 and the final pointer referencing a node greater than 16.

If a key is inserted into a node which is already full it causes overflow, which results in the node splitting into two and the key with the middle value being chosen to become the new parent of these two split nodes. This new parent is then inserted into the parent node of the split nodes. If insertion into the parent node itself causes overflow then this process continues until

either overflow stops occurring or the root node is reached. If the root node overflows it is split and a new root is created.

There are several variants of B-Tree, including the B+ Tree which only stores data at the leaf level. This offers the advantage of being able to store more keys per block (which reduces the number of disk reads) and, because all keys are represented at the leaf level blocks can be linked together, which makes sequential reads of the data extremely easy and efficient.

2.5.3.2 R-Trees

B-Trees and their variants are designed to represent and perform queries on one-dimensional data. R-Trees [46], on the other hand were designed to represent the geometric data (e.g. points, lines, surfaces and volumes) of any N-dimensional space, and to perform queries on this data. The geometric data is represented within the tree using it's Minimum Bounding Box (MBR) which refers to the smallest N-dimensional rectangle which can completely enclose the geometry. This greatly simplifies the search algorithm. As with B+ trees, R-Trees store all of their data objects in the leaf nodes only, with intermediate nodes being responsible for storing MBRs and pointers to child nodes.

Node overflow is handled similarly to the way it is handled by B-Trees, except that the split procedure is more complicated as it aims to create two nodes which minimise the total MBR area rather than split based on position. Another difference is that the split must be registered on all parent nodes up to and including the root to ensure that their MBRs are of the correct size.

A number of topological relationships have been defined between the query and data object MBRs which allow different types of queries to be performed on the tree [58]. These include:

- Disjointness - The two MBRs do not meet.
- Meets - The two MBRs meet at one edge.
- Covers - The two MBRs are fully intersected and meet at one edge.
- Overlaps - The two MBRs are partially intersected.
- Contains - The two MBRs are fully intersected.
- Equals - The two MBRs are fully intersected with all edges of MBR A meeting the respective edges of MBR B.

Queries begin at the root node and descend down the tree, entering any nodes whose MBR satisfies the topological relationship specified. Traversal continues until the leaf level is reached. At this point, if any objects are still under consideration the query is performed on the actual geometry to determine whether it actually matches the object or only matches its MBR.

A variety of query types are available, many of which are summarised below:

Window Queries Obtain all objects which are stored within a rectangular range.

Nearest Neighbour Queries Obtain the k nearest objects from a given point. Incremental Nearest Neighbour returns the data objects in the order that they are found.

Spatial Join Queries Obtain the objects which satisfy a spatial predicate formed of two or more topological relations combined using conjunctions and disjunctions.

As with B-Trees, an R* variant of the R-Tree algorithm was created which offers much improved performance. The R* tree split functionality is based on the observation that the gradual insertion of keys over time likely leads to a sub-optimal tree, with bulk loading (the insertion of all keys "at once") producing more efficient results. Although it is obviously not always possible to insert all keys into the tree at once, the R* tree derives some of its benefits by re-inserting a fraction of the keys whenever a split occurs. This helps to achieve a better structure than could be achieved by splitting alone. The algorithm also places some emphasis on the elimination of overlapping keys as a large amount of overlap results in many branches of the tree having to be explored which are in all likelihood fruitless.

2.5.3.3 Hilbert R-Trees

The Hilbert curve is a space-filling curve [54] which makes it possible to construct a mapping from 2-dimensional data to a 1-dimensional representation which preserves data locality.

The Hilbert R-Tree [54] represents geometric objects using the Hilbert value of their centroid. In addition to the MBR, each internal node also stores the maximum Hilbert value of the subtree. When inserting a new rectangle its centroid is calculated and compared against the Hilbert values of each potential subtree at each node. The subtree with the smallest Hilbert value larger than the new value is chosen.

One advantage of the Hilbert R-Tree is that the inclusion of the Hilbert value at each node ensures that there exists an order to nodes which allows siblings to be used as storage when overflow occurs. Thus, splits are only necessary if all siblings are also full which can help maximise storage utilisation.

Hilbert values can be extended to N-dimensional data but it has been shown that performance degenerates when using a large number of dimensions ([14], pg. 144). This degeneration or "dimensionality curse" [58] is unfortunately found in most spatial trees and is a consequence of excessive MBR overlaps.

Trees which are designed to combat this degeneration are discussed in the section 2.5.3.5.

2.5.3.4 Distributed R-Trees

Bianchi [11] proposed a Distributed R-Tree (DR-Tree) intended for use in publish-subscribe applications. They evaluated their algorithm as applied to R*-Trees and Hilbert R-Trees. Subscribers insert a poly-space rectangle representing their subscription into the tree and publishers do the same for any content they wish to distribute. Published content traverses the tree until it is delivered to all subscribers. Subscription and publication times are logarithmic in the size of the network and each node has a polylogarithmic memory requirement. The tree is fully distributed and self-organising.

The connections between physical nodes (p-nodes) are dictated by the semantic relationships between subscriptions. That is, p-nodes are connected based on their location within the tree, with each node holding the addresses of between m and M child nodes and one parent node. The child nodes are responsible for more specific regions of the tree whereas the parent will be responsible for a broader region.

To ensure balance, each node may be responsible for a virtual node (v-node) at several contiguous levels of the tree from the leaf upwards. The number of v-nodes a given p-node is responsible for depends on the number and structure of the subscriptions entered as a new v-node must be created whenever a node split occurs to represent the newly created node responsible for the split nodes. Each p-node has to maintain parent and child sets for each of their v-nodes.

The R-Tree structure guarantees no false negatives (i.e. all subscribers will receive any relevant publications) and a small number of false positives. To ensure a low number of false positives two containment properties should be maintained between nodes (with $\sqsubseteq$ representing containment):

- If A $\sqsubseteq$ B then A cannot be an ancestor of node B.
- Either
    1. If A $\sqsubseteq$ B then B is an ancestor or sibling of A.
    2. If A $\sqsubseteq$ B, A $\sqsubseteq$ C, B $\not\sqsubseteq$ C, C $\not\sqsubseteq$ B then A may be a descendent of either B or C (when a node has two containers).

A node joins the network using an oracle which is assumed to produce a node in the network. To ensure good insertion in the tree the join request is propagated recursively up to the root node where insertion begins.

Each node must maintain a filter representing their subscription and a child set, parent set and MBR for each level of the tree that it is present within. Note that the MBR represents the minimum bounding rectangle of the MBRs' of all children at level L, and these sets must be maintained as the tree evolves.

Dynamic reorganisation and self-stabilisation is achieved by periodically running algorithms to check tree structure and to make repairs if the structure is in an invalid state as a result of transient faults such as dropped messages or node failure. The system periodically checks for the correctness of MBRs, node covers, child/parent sets and the child size invariants (i.e. that the number

of children range between m, and M). Issues are dealt with by correcting any broken invariants and then propagating the changes up the tree if possible, or via reinsertions and splits if necessary.

Message dissemination can be made more reliable by buffering messages in nodes and asking for retransmissions when the network is stable. The degree to which this is possible depends on the amount of available memory, and the length of time that it is desirable to buffer the data depends on the data stored. For example, if dealing with the dissemination of stock quotes, messages should only be buffered for a short period (if at all) due to their transient nature.

Valero et al. [87] note that the DR-Tree relies on the reinsertion of nodes when a subtree is disconnected. This can result in high message traffic and long stabilisation times. They proposed a modified DR-Tree which replicates each non-leaf node on any node in the network which holds one of its children. When it is detected that node P has failed, every p-node holding the leftmost replica of each v-node held by P restores the v-node. This results in a change to the p-node interaction graph but does not alter the tree structure. It also has the desirable property that restoration only concerns those peers containing a replica of P.

Bianchi's DR-Tree provides spatial decoupling because each node is only required to know of a small number of other nodes in the network but is still able to subscribe to and receive publications from any node. Temporal decoupling is supported provided publications are buffered at nodes within the tree for later transmission to new and recovering subscribers. Finally, synchronisation decoupling is provided as each node will be notified when new publications arrive that match their subscription request.

2.5.3.5 Supporting Multiple Dimensions

The TV-tree [56] is designed to support high dimensional data by only taking into account the dimensions at each node which are not shared by all keys, thus eliminating from consideration those dimensions which cannot make a difference to filtering.

The X-Tree [10] is another alternative which acknowledges that in high dimensions a high amount of overlap is likely to occur. This will be very inefficient as it means that a large proportion of the branches will have to be traversed. The X-Tree makes use of hierarchical storage (identical to the R-Tree) for keys with little overlap but uses sequential storage in cases where the overlap is high. Sequential storage in these cases is more efficient as it allows those keys which will need to be read anyway to be stored sequentially in memory rather than randomly. This approach is only used if attempts to reduce overlap are unsuccessful.

The X-Tree shows itself to work extremely well for dimensions over two, with speed increases of up to 450 as compared to the R*-Tree and between 4

and 12 times for the TV-Tree. It also exhibits insertion time improvements of 8 and 30 times for the R* tree and TV-Tree respectively.

Our middleware stores each Contract supported by an application as a single dimension, so a spatial index is an appropriate data structure to use for indexing messages. Nodes are also able to lookup and bind to their n most geographically proximate neighbours. Again, a spatial index is appropriate as it allows us to store the locations of nodes and offers nearest neighbour query functionality.

In addition, a distributed R-Tree algorithm is available for use, and a replication scheme has been proposed and evaluated which allows us to ensure the fault tolerance of the index.

2.6 Summary

In the preceding chapter we have provided an overview of various distributed protocols using the notions of spatial, temporal and synchronisation coupling to illustrate how they each exhibit different properties and capabilities; and that care should be taken when building any distributed application to ensure that the chosen protocol lends itself well to the types of data and communication patterns used.

For our middleware we have chosen to use the tuple space paradigm for its temporal decoupling and straightforward communication interface, and a distributed spatial index for message and node lookup because of its support for multiple dimensions and its nearest neighbour query functionality.

Now that we have discussed the applicable protocols for our middleware we move onto a discussion of context-awareness. Specifically, we define context, discuss a number of ways of representing it and evaluate several context-aware frameworks from the literature. We evaluate these frameworks in terms of four criteria used during the design of our middleware.

3 Context-Aware Middleware

3.1 Introduction

The generally accepted definition of context is stated by Dey and Abowd [2] as "any information that can be used to characterise the situation of an entity, where an entity can be a person, place or physical or computational object". More specifically, they defined a place as a geographical space (e.g. room, office, buildings, streets) and further divided people into individuals or groups (which may be co-located or distributed). They further stipulated that these entities can themselves be defined along a number of properties [28] which we shall now briefly discuss:

Identity The ability to assign unique ID's to every possible entity in the application area.

Location In addition to the (X, Y) co-ordinates of the entity, this also refers to orientation, elevation and spatial relationships between entities such as co-location, proximity and containment.

Status Refers to characteristics of an entity that can be sensed. These vary depending on entity type; with examples including temperature and light level for place, mood and activity for people (taken either for individuals or groups) and CPU load or application state for physical and computational objects.

Time Although potentially useful on it's own, time is often used in conjunction with other types of context to form a history by associating it with timestamps or ranges. This allows the system to derive the relative order of events or causality.

Context awareness refers to the situation where software or devices are aware of their situation and can adapt appropriately without user intervention [19].

Middleware in the context of this research refers to "software that mediates between an application program and a network. It manages the interaction between disparate applications across the heterogeneous computing platforms" [1]. A common example is the Object Request Broker (ORB) which is part of the CORBA specification[1] and allows applications to invoke methods on remote machines. These invocations can be made programming language agnostic (e.g. you may call a C++ method from a Java application) through the use of the CORBA Interface Definition Language (IDL).

One of the first examples of a context-aware system was the Active Badge Location System by Want et al. [89]. They designed a system to track the location of individuals within a building using wearable badges equipped with an infra-red emitter which sends a signal to a network of sensors placed around the building once every 15 seconds. These signals were aggregated at a master computer and made available to clients to display visually. The badges also contained a light sensor which turned it off when in the dark.

Another pioneering study was that of the GUIDE context-aware electronic tourist guide conducted by Cheverst et al. at Lancaster University [20]. They observed that group-based tours of a location are "inherently inflexible" because of the need to target the majority at the expense of individual interests. To solve this issue they developed a computer-based tour guide with an awareness of both location and user interests. This allowed them to deliver information related to nearby landmarks and to tailor this information based on registered interests. Tours could be created by users listing all the attractions they wished to visit, which were then intelligently scheduled by the system. Tour scheduling could be updated dynamically based on new information (e.g. users staying in one location longer than expected). Location was determined by receiving broadcasts from base stations placed at important locations throughout the city. These base stations contained state pertaining to the location they represented (e.g. opening times) which could be delivered to the user and used in scheduling.

3.2 Context-Aware Frameworks

In order for a device or application to reason over its context it will need to observe its environment and internal state. It will then need to reason over this context to decide on an appropriate action. The external environment can be observed through the use of sensors and a variety of technologies can be used to reason over the sensor data to make a decision. Internal state can be specified by the user or derived through data-mining.

Although all context-aware applications are different, it is possible to derive a set of common behaviours and requirements of all systems. To stop developers from having to keep "reinventing the wheel" a number of middleware

[1] http://www.corba.org/

solutions have been proposed which perform these common tasks and offer outputs to the developer via well defined APIs.

Over the last decade a significant number of frameworks have appeared which abstract the sensor reading and evaluation processes to the use of small APIs and context models [83] (e.g. ontologies, key-value, graphical). These systems have gradually increased in sophistication; from providing only simple abstraction and inference mechanisms (such as being able to signal when an individual leaves a room) [78] to systems providing more powerful inference mechanisms [18, 55] through specialised languages (such as OWL[2]) and conditional structures (e.g. $\wedge$, $\vee$, $\exists$) [45].

Many of the systems discussed have been designed to be either centralised or localised (requiring direct communication with a server tasked with aggregating and processing the data). For small-scale or localised networks these systems may be sufficient. However, they are largely inadequate for the construction of large scale networks. Many researchers have noted this and have taken it into account in their designs. These wide-area frameworks also range in sophistication. Some frameworks simply provide the sharing of sensor data between devices via short-range communication (e.g. BlueTooth) [50], whilst others combine local and infrastructure communication to support more powerful applications [79]. Both localised and wide-area systems are considered in sections 3.9.1 and 3.9.2 respectively.

3.3 Categories of Context-Awareness

Chalmers et al. [16] suggest that the uses of contextual information fall into six main categories:

Context Display Displaying the context information to the user. Common examples on smart phones include location and weather information.

Contextual Augmentation Annotates data with the context of their creation. For example, diary applications sometimes allow the user to annotate a diary entry with location and weather information.

Context-aware configuration For example, a device may decide where to print a document based on it's proximity to a set of printers [16].

Context-triggered actions For example, a device may notify it's user if rain is forecast for the following day.

Contextual mediation Using context to best meet the needs and limitations of the user and their device by modifying a service or the data they receive. For example, a service may dynamically change the quality of graphics being transmitted to the device depending on bandwidth.

[2] http://www.w3.org/TR/owl-features/

Context-aware presentation Adapts the presentation of data according to user or device context. For example, many applications alter their user interface depending on whether they are running on a smart phone or larger tablet.

They apply these last three categories to improve the visual information displayed on an in-car satellite navigation system [15]. For instance, context triggered actions are used to load map data based on the user's predicted next location and context-aware mediation and presentation are applied to the map which shows more or less detail depending on the speed that the car is travelling.

3.4 Aspects of Context

Context comes in many forms, each of which needs to be handled differently by a system. Haghighi et al. [48] distinguish between dynamic context (which changes continuously like location) and static context (e.g. username).

Environmental context, usually derived through the use of sensors and personal information, is either specified explicitly by the user or derived implicitly; for example by data-mining a user's E-Mail inbox. Gellersen et al. [40] propose the additional distinction of direct and indirect awareness. Direct awareness refers to context obtained on the device itself (e.g. location information via a local GPS sensor) whereas indirect awareness refers to context obtained through the use of some external sensor or service (e.g. location through the use of RFID beacons).

Context can also be percieved at different levels of abstraction as noted in [50], who distinguish between physical and logical context. Physical context represents the raw sensor data whereas logical context refers to more abstract higher-level information. The example given is that of location; with GPS co-ordinates representing the physical context and street names representing logical context.

Ever higher levels of abstraction can be achieved by combining the data of several sensors or logical contexts to form composites. For instance, Fahy and Clarke [34] point out that in order to derive weather predictions, many types of sensor data are necessary including temperature, light-level, rain, wind, humidity and barometric pressure.

The efficacy of context information is often improved greatly when you observe a sequence of readings instead of just each reading as it occurs (i.e. historical context information). Through the use of this history (often taken from several sensors) it is possible to derive highly abstract contexts such as "running" or "sitting down" [64].

Many specific examples of context are enumerated in the literature [17]. These include:

Networking Context network connectivity, communication costs, bandwidth, nearby resources (e.g. printers, displays, other devices)

Device Context screen size/resolution, supported codecs [64], available sensors

User Context profile, location, nearby people and objects, social situation, focus of attention, orientation [5]

Physical Context Lighting, noise levels, traffic conditions, temperature

Time time of day, week, month, year or season

3.5 Representing Context

Context information at it's most basic level corresponds to the triple: *(entity, contextType, sensorValue)* where entity refers to the person, place or thing the context value represents. However, this representation is often inconvenient for defining context conditions so abstract representations are used. Probably the simplest abstraction is that of the numeric range; where it is specified that an aspect of context should fall between two values (e.g. 25 - 50).

A significantly more powerful representation is that of the ontology. Ontologies can be simply used to map value ranges to descriptive labels known as concepts (e.g. 0-6 representing cold) or they can be used to specify complex relationships between entities with additional semantics (e.g. reflexivity or transitivity). Hierarchical ontologies are used [16] to model generalisations and specialisations of concepts, giving it many of the advantages inherent in object-orientation.

It should be noted that sensors are often noisy and error-prone so the quality of the context produced is uncertain [16]. Chalmers et al. [16] suggest that this issue can be alleviated by mapping a context value to a range of possible values, with the size of the range depending on an application defined confidence value (between 0 and 1). The confidence value represents how certain the context-aware system is that the correct value for a given context is within the specified range. Therefore, larger confidence values tend to correspond to larger ranges and vice versa. They define a mapping strategy for numeric and hierarchical ontologies and propose a number of relationships that can be defined over uncertain context allowing the system to make more informed decisions.

3.6 Context Models

We argue that the choice of context model is a very important consideration when designing a framework as the choice of model dictates the types of

relationships which may be expressed and the forms of reasoning which may be used. We shall now briefly discuss many of the various context models and attempt to illustrate their properties [83].

While early models (such as [78]) used simple representations such as key-value, these have now been largely abandoned in favour of more expressive forms (such as ontologies, object-orientation and logic models) which support complex relationships between entities and reasoning capabilities. [48] argue that SQL and ontology based models are the best suited for representing and reasoning over all aspects of context.

3.6.1 Key-Value Pairs

Although easy to parse due to their simplicity, this model lacks any expressivity, usually only being capable of performing exact match reasoning. This makes it impossible to express non-trivial relationships between data and hence is only useful for the simplest of context queries.

3.6.2 Markup-Based

These models are all based on the SGML markup standard[3], and more specifically are usually expressed in XML[4]. XML makes use of pairs of tags describing the data they surround. This simple definition makes it approximately as flexible and expressive as simple key-value pairs. However, their power comes from their support of tag hierarchies and properties which allow for more sophisticated modelling. For instance, the CC/PP[5] and UAProf[6] vocabularies allow you to describe device capabilities and user preferences in order to customise incoming data from servers or other devices. The CSCP model takes advantage of tag hierarchies to support context-sensitive naming so that tags and property names do not need to be unique across the entire profile [83].

Markup-based models are also capable of expressing extremely complex contexts and any relationships between them; as exemplified by the XML representations of ontology languages such as RDF[7] and OWL[8] (discussed later).

3.6.3 Graphical

Graphical models such as UML and ORM diagrams allow for relationships and dependencies to be expressed. Henricksen et al. [49] proposed an extended

[3] http://www.w3.org/MarkUp/SGML/
[4] http://www.w3.org/XMLowl/
[5] http://www.w3.org/Mobile/CCPP
[6] http://www.wapforum.org
[7] http://www.w3.org/RDF/
[8] http://www.w3.org/standards/techs/owl

ORM model supporting both static and dynamic data, with dynamic data being further categorised as profiled, sensed, derived, or temporal. The model allows you to make inferences given facts (e.g. given that person A and B are located in room C, it can be derived that person A and B are in the same room) and to express dependencies between facts (if fact A changes, fact B must change also).

Graphical languages intended as an alternative to SQL have also been proposed [48] such as QBE (Query-By-Example) that generates queries based on example tables constructed by the user. Other systems such as Chiromancer [67] extend the QBE paradigm by allowing users to construct queries on handheld devices using familiar concepts from the desktop user interface paradigm.

The Bigraph model discussed briefly in Section 3.7.1.2 has a well developed graphical representation to accompany its formal calculus-based approach [63].

Graphical models have the advantage that they are more human-readable than other approaches and can potentially be used by those who are less technically experienced. Although graphical representations can be constructed for non-graphical models it is unlikely that they would be as easy to comprehend because graphical display was not a central consideration during design.

3.6.4 Object-Orientation

The main benefits of using an object-oriented modelling approach are the possibilities for encapsulation and reuse. Regardless of the dominant model used, practically all systems make use of these principles in some form. For example, the complexity of sensor handling (fusion, normalisation, noise elimination etc) may be encapsulated within an object, with all communication being performed via a small interface.

Object-Oriented models can be represented graphically using UML and a number of other representations.

3.6.5 Logic-Based

This approach models context in terms of facts, rules and expressions. The main benefits of this model are the formality of the representation and the powerful inference capabilities present within the logic interpreter. It also has the benefit of allowing additions to the model to be made rapidly simply by adding additional rules and allows the additional or removal of context information through the additional or removal of facts from the model. Gaia uses a logic-based model for their context infrastructure [71].

3.6.6 SQL-Based

SQL (Structured Query Language) is a declarative programming language used primarily for querying relational databases. This model has been applied to the querying of context by a number of systems, either by using it as inspiration in the design of a context-aware variant of the language (as with Coalition [19], discussed later) or by providing a mapping from a context-aware language to an SQL database [62].

3.6.7 Ontologies

Ontologies are used to model a domain in terms of classes and subclasses of objects, properties held by said classes and the relationships that exist between classes. Their major strengths include their inference mechanisms and the fact that ontologies may be shared freely and incorporated into other ontologies with ease. An example scenario could be of a system which requires a certain type of context unavailable for some reason (such as the failure of the node holding it) An ontology reasoner could infer whether any equivalent contexts are available and use one of them instead.

There have been a number of ontology languages proposed over the years [53], used primarily in the fields of artificial intelligence and the semantic web. One of the most developed and flexible of these is OWL, which has its roots in description logics and the earlier languages RDF, RDFS[9] and DAML + OIL[10]. OWL is semantically equivalent to well-known description logics which allows inferences to be made on the data through the application of a reasoner.

However, the act of performing inferences over an ontology is expensive both in terms of system resources and time. For this reason (amongst others) OWL was specified as three separate but linked languages [53]:

OWL DL Decidable inference and equivalent to the description logic SHOIN. It has a worst-case nondeterministic exponential time (NEXPTIME) complexity with no "practical" complete algorithm for inference.

OWL Lite More tractable inference than OWL DL with a worst-case deterministic exponential time complexity (EXPTIME) and practical optimised algorithms for inference. However, it has limited cardinality support and lacks the union, intersection and complement boolean combinations

OWL Full Undecidable but more expressive than OWL Lite and OWL DL, supporting all the features of RDFS.

[9] http://www.w3.org/TR/rdf-schema/
[10] http://www.daml.org/

3.7 Location Models and Services

We now discuss location. This is arguably the most important type of context so we devote the following two sections to a discussion of location models and services. Location models provide a means of representing the location of entities and any relationships which exists between them. Location services are distributed applications which allow entities to share location information about themselves and others. The ability to determine the location of users and computer systems is extremely important to context-aware applications, and in particular to pocket-switch networks and any network which uses location explicitly as part of the routing process.

3.7.1 Location Models

Location models in their simplest form provide us with a means of representing physical locations as values we can use within applications. Location models vary in levels of detail and scope and may be global (representing location across the world) or local (e.g. representing location within a building or set of buildings).

Locations are represented using a co-ordinate system which may be defined as a set X of coordinates, where a coordinate is "an identifier which specifies the position of an object with respect to a given coordinate system." [9]. Probably the most ubiquitous coordinate system in used today is WGS84 which is used by GPS [9].

Becker and Dürr [9] argue that in general a location model should provide:

- Position,
- A distance function for performing distance-based calculations such as nearest neighbour,
- A notion of connectedness to allow navigation (e.g. connections to indicate a door connecting two rooms) and,
- Containment relationships to allow range queries (e.g. locating all rooms within a building).

They may also support orientation (e.g. so that a system can establish where a user is looking).

Locations models can be divided into two broad categories: geometric and symbolic.

3.7.1.1 Geometric Models

Geometric models (such as GPS) use geometric coordinates. These can be used to refer to "a point or geometric figure in a multi-dimensional space" [9] (usually a plane or three-dimensional space). Because of their basis in geometry, these models allow the definition of a distance function and other

topological relationships such as overlap and containment. They do not however support all topological relationships (such as connectedness). These need to be defined explicitly within the model.

3.7.1.2 Symbolic Models

Symbolic models are represented using abstract symbols such as street names or sensor IDs. These symbols can be structured in a number of ways - each of which providing different capabilities. Four such structures are set-based, hierarchical, graph-based and Bigraph-based. Each of these will now be discussed:

Set-Based Represented as a set L which contains all of the symbolic coordinates of the model. Locations can be represented as subsets of L. For example, if L contains all of the room numbers in a building you can divide the building into floors by constructing subsets containing all of the rooms of each floor. Overlap between two locations $L1$ and $L2$ can be established when their intersection is non-empty ($L1 \cap L2 \neq \emptyset$) and $L1$ is said to be contained within $L2$ when $L1 \cap L2 = L1$. Connectedness can be represented as sets of directly connected locations and larger neighbourhoods can be defined by recursively joining smaller neighbourhoods with non-empty intersections. Set-based models have no notion of distance.

Hierarchical Models Represented as a set of locations L ordered according to their spatial containment relationships. $L1$ is an ancestor of $L2$ ($L1 > L2$) if $L2$ is spatially contained by $L1$. This set can be represented as a tree, or if overlap is an issue, a lattice can be used with location intersections being modelled by separate locations with more than one parent. The hierarchy is used to support range queries where descendants of a location L are within the range of L. Connectedness cannot be represented using a hierarchical model.

Graph-Based Represented as a graph with edges between vertices representing a direct connection between two locations. Edges can be weighted to model distance and range queries can be applied by specifying a reference vertex with a radius to specify the range. This model does not support the building of larger neighbourhoods through the recursive joining of smaller neighbourhoods.

Bigraph-Based Although not limited to representing location, Bigraphs can be used for this purpose, allowing the modelling of containment and connectedness. It has the additional benefit of allowing the definition of reaction rules which precisely specify actions that may be performed. For example, a rule may be defined to represent a user leaving a room or connecting to a computer [63].

3.7.1.3 Hybrid Models

Location models can be combined to incorporate the benefits of each into a single model. Set-based and graph-based approaches can be combined by representing locations as sets and using graphs to place connections between locations. This hybrid model supports range queries, connectedness and distance calculations and has the added benefit of making it straightforward to provide a representation at different levels of granularity (e.g. individual rooms, floors or entire buildings). Graphs and hierarchy based approaches can also be combined to support containment, overlap, distance calculations and connectedness.

By combining geometric and symbolic approaches you can achieve greater accuracy and precision for distance queries. You can also use arbitrary geometric figures for performing range queries and containment checks. Geometric information may be provided for every modelled location or only for some. For example, a coordinate system may be used to represent buildings but not rooms, with geometric values being approximated for unsupported locations. In cases where only a partial geometric model is provided either a top-down or bottom-up approach is often taken. In a top-down approach, the root location (e.g. the entire world or a building) has exact values along some coordinate system but the values of descendent locations can only be approximated. The bottom-up approach is the opposite of this, with exact geometric values for the most specific locations (such as a room) and only approximate values as you ascend towards the root.

3.7.2 Location Services

Location services allow you to establish the locations of other nodes in the network. Knowledge of the geographical location of other nodes can be useful for a number of reasons. For instance, a device could use the information to connect to the closest server when downloading files or could share geographically sensitive contextual information with a nearby device.

Landmark routing is an example of an early location algorithm which uses router information to reduce the distance travelled when sending data to a destination. Landmark routing has been applied in services such as the topology-aware overlay devised by Xu et al. [93], which also makes use of round-trip-time to refine location and increase service efficiency.

Mauve and Widmer [61] provide a detailed account of many location services and describe them in terms of the responsibilities of nodes in the network. Specifically, they highlight the number of nodes hosting the service (*all* or *some*) and the amount of information stored at each service node (storing location either for *all* or *some*). These properties are summarised as *some-for-some*, *some-for-all*, *all-for-some* or *all-for-all*. They also discuss how the systems vary in terms of communication complexity (the amount of commu-

nication between nodes), position accuracy, robustness and implementation complexity.

The Dream service uses an *all-for-all* approach where all nodes store the location information of all others. Each node regularly uses restricted flooding to update their position at other nodes, with the update being discarded after travelling a specified distance. This approach ensures that a node maintains up-to-date information about other nearby nodes but only holds approximate locations for more distant nodes. It is noted that the frequency of updates should be proportional with the speed of travel of the node because the faster the node is travelling, the more pertinent the information. Although the service is highly robust and easy to implement it has high storage needs and communication complexity.

Homezone is an *all-for-some* approach where each node maintains a conceptual disk around itself with radius R and is positioned at a centre point C. The position of C can be obtained by applying a known hash function to the node identifier. All nodes within the radius of a node store its location information. Homezone has low implementation complexity and good communication complexity but the radius of nodes may need to be enlarged in sparse networks to ensure good operation.

Quorums have been used extensively to provide replication in databases and distributed systems and have been used successfully in a number of location services [61]. Quorum based systems are usually *some-for-some* approaches and have been shown to be efficient, robust and space conserving, but with a high implementation complexity. A quorum consists of two subsets of the network whose intersection is non-empty. One subset is used for receiving information updates (write operations) from the network and the other is used for handling information requests (read operations). Quorums communicate via some non-position based routing scheme. It can be shown that using a quorum-based approach ensures that an up-to-date version of some node's location can always be found.

The Grid Location System (GLS) divides the geographical area into a hierarchy of squares using a quad tree [61], with each level of the tree further dividing the space into four smaller areas. Each node maintains a table of the position information for all other nodes within the local first-order square and location lookups are handled by progressively traversing through the network, and contacting the neighbour node closest to the destination until the destination is found. GLS is efficient, space sensitive and robust with medium implementation complexity.

GLS usually performs better than Homezone for mobile nodes while Homezone is superior if nodes tend to be close together.

3.8 Criteria for Evaluating Existing Frameworks

With the overall goal (providing a decentralised context-centric network) in mind, each of the discussed frameworks are now compared along the four scales deemed important:

Flexible Evaluation Refers to the extent that the system may be used to form complex contextual conditions (for example, allowing the use of logical connectors such as conjunctions, disjunctions and negation) and the extent to which the systems are capable of reasoning over contextual information (i.e. how effectively more complex contexts may be derived from simpler contexts).

Extension Refers to the ease at which models of context may be added to or modified by developers. The ideal scenario would be to allow alteration of the ontologies using some high level notation or graphical environment which does not require modification of any of the sensor reading code.

Heterogeneous Interoperability Refers to the extent that the system restricts the choice of programming language which may be used. For example, if a system provides its callbacks over Java RMI the developer would be restricted to using the Java language (without considerable extra effort). Use of a protocol such as SOAP[11] however would be far less restrictive.

Centralisation Refers to the degree that systems rely on either a centralised or localised infrastructure for storing and retrieving context information.

Centralised Systems refer to those which use a central server (or collection of servers) to handle the requests and inputs of all clients. Within this infrastructure almost all data is stored on the server machines.

Localised Systems are almost identical to centralised systems but with the difference that access to the system is only available over a reasonably small geographic area.

Reliance on these infrastructures can be undesirable. Centralised networks may be criticised for both monetary and logistical reasons (such as the cost of purchasing and maintaining server machines) but their main limitations are their lack of support for internet incapable devices and scalability issues where as the number of devices in the network grows the centralised servers become a "bottleneck" which can seriously reduce system throughput. This problem can be alleviated by purchasing additional machines but this causes further problems such as making data replication tasks more expensive.

[11] http://www.w3.org/TR/soap/

These criticisms are less applicable when discussing localised networks; however they instead suffer from their lack of applicability to the wider world - being of use only within the small area it covers.

3.9 Evaluating Frameworks

As discussed above, context-aware frameworks can be broadly categorised according to whether they have a localised or decentralised architecture. We have chosen to structure the evaluation which follows using this categorisation.

3.9.1 Localised Frameworks

The *Context Toolkit* by Salber et al. [78] takes inspiration from graphical user interface (GUI) "widget" libraries. These libraries provide all of the functionality for handling GUI components such as text fields and combo boxes other than the application specific logic. The context toolkit emulates this by encapsulating the entirety of the sensor reading code within a sensor widget containing various "attributes" (such as last sensor value and sensor location) and callback functions which trigger on important events occurring. The system has good support for interoperability as communication is handled via HTTP and an XML language. However, it lacks any native means of performing flexible evaluation (partially due to the choice of a simple key-value model which is incapable of expressing relationships between the data) and makes extension difficult as each ontology is tightly coupled with the sensor reading code.

The *CoBra* ontology system [18] and The *Context Management Framework* [55] both provide much more sophisticated ontologies, allowing higher levels of context to be inferred. In addition, they both use XML to represent their ontologies which makes extension easier and provides good interoperability.

CoBra is defined using OWL and determines the current context through the use of a knowledge base queried by an inference engine. It allows the definition of far richer semantic information about entities (e.g. it can be stated that a location subsumes another) which allows more powerful inferences to be made; and uses standardised XML protocols (such as SOAP) for communication.

The *Context Management Framework (CMF)* defines ontologies in terms of a hierarchy (e.g. Environment:Sound:Intensity) and improves upon previously discussed models in two main ways. Firstly, it allows the specification of higher level ontologies by combining the values of two or more lower level ontologies. An example could be the combination of several sound based ontologies (Harmonicity, SpectralSpread, Transients etc) to form a higher level context of SoundType (Car, Elevator, RockMusic etc). Secondly, the model

accounts for imperfect or partially ambiguous sensor data by allowing for ontologies to be modelled using fuzzy sets and uses a Bayes probability model [41] to make inferences about the context. The framework also provides much of the desired evaluation flexibility (AND, OR, NOT).

SOCAM [45] provides excellent evaluative capabilities, allowing the use of logical connectives (e.g. $\wedge$, $\vee$, $\neg$), quantifiers (e.g. $\exists$) and the capabilities provided by RDF/OWL (e.g. transitivity). Uncertainty is handled through the use of a Bayes probability network. The use of OWL ensures both extendability and interoperability.

3.9.2 Decentralised Frameworks

One of the first frameworks to support decentralisation was *Hydrogen* [50], where devices communicated via short range protocols (such as BlueTooth) to share sensor readings. This was an important first step, but for all it gains in decentralisation it unfortunately loses in flexibility and extendability.

More recently researchers have been looking into the feasibility of wide-area sensor frameworks. To this end, many have studied the combination of server machines with pocket switch and related ad-hoc networking solutions. One example of this approach is in the work of Santa and Gómez-Skarmeta [79] which provided car users with useful information (e.g. traffic reports, tourism and travel information, pollution problems etc) through the combination of *Vehicular Ad-Hoc Networks* (providing vehicle-to-vehicle and limited vehicle-to-infrastructure communication) and cellular networks (providing vehicle-to-infrastructure and infrastructure-to-vehicle communication).

Eisenman et al. have gone further, ambitiously proposing that large scale sensing can best be served through the addition of a new wireless sensor edge for the current Internet [32]. They emphasise three key principles which should be followed for success:

Network Symbiosis We should harness existing knowledge and technologies as much as possible. For instance, use should be made of wireless access points, existing routing protocols and security methodologies.

Asymmetric Design The authors acknowledge that systems vary in capability (bandwidth, processing power, battery capacity etc) and suggest that protocols should leverage this asymmetry to provide a better service; for example, by pushing computationally intensive tasks onto more capable nodes.

Localised Interaction They argue for the communication range of all network nodes to be heavily constrained to be within "spheres of interaction" (e.g. radio range) - with the motivation being that this will facilitate simplified design and communication performance.

> These three principles were followed throughout the development of their *MetroSense* architecture. Of particular note is the use of a three tiered architecture (Server, Sensor Access Point (SAP) and the Sensor Tier), with each tier being allocated appropriate capabilities and being built upon current infrastructure including generic server machines and wireless access points (for SAPs'). The (perceived) limitation of localised interaction is overcome through the process of opportunistic delegation, which is the process of delegating jobs to nodes as they are encountered (e.g. to collect sensor data). These nodes may themselves delegate and so on. In this way, a node can obtain or send data across great distances while still retaining the benefits of localised interaction.

Coalition [19] is a context processing framework which allocates processing to other nodes in the network rather than rely on a centralised system. The system does however use a centralised "Manager" to co-ordinate all activity.

Their rationale was that by distributing the processing they could avoid a bottleneck at the Manager and achieve better overall throughput.

Coalition performs context-based queries using a Context Query Language (CQL) which operates on attributes of context domains. For example, the context domain "Person" may have an attribute "preference". CQL is structurally very similar to SQL.

Distributed context processing is achieved through the use of a generic representation of the context query information called a "Query Plan" created by the Manager. The manager parses the CQL and produces this generic representation which contains a list of the context information to be retrieved and methods for performing this retrieval. Each Query Plan also contains the address of the requesting node so that the processing node can interact with it without needing to involve the Manager. The other main benefit of the Query Plan is that it eliminates the need for the processing nodes to understand the CQL.

The Manager dispatches the Query Plan to the requesting node for processing, and also dispatches it to a random node within the group of nodes that can satisfy at least some of the context information required. This random node then uses a peer-to-peer network to dispatch the plan to all other related nodes. The context information is obtained and reported back to the requesting node, which then processes the query and returns it's results to the application level program.

The authors compared *Coalition* against it's original centralised form, finding that this decentralised approach greatly reduced average query time and increased system throughput.

The design of *TOTAM* (Tuples On The AMbient) [13] solves an issue within all previous federation or replication based distributed tuple spaces; that is, the possibility that incorrect context information is perceived.

```
totam.inject (tuple: [VirtualObject, grenade, location]);
inContext: [tuple: [TeamInfo, ?u, GANGSTER],
            tuple: [PlayerInfo, ?u, GANGSTER], ?loc],
            inRange(location, ?loc) ]
```

Table 3.1: An example of a TOTAM context rule [13]

In the case of federation based sharing this issue is due to the inaccurate ranges of sensors. For example, the range of a context provider broadcasting a location will likely be smaller or larger than the location it represents. Thus, some nodes may erroneously detect that they are present in a location when they are not (if the context provider range is larger than the room); and some nodes may erroneously detect that they are not present in a location when they in fact are (if the context provider range is smaller than the room).

Using the location example again, the issue with replication based services is that a node (A) outside the room may come into transmission distance with a node (B) inside the room. This would result in the location tuple being incorrectly replicated to A.

The *TOTAM* middleware uses the Tuple Space paradigm to support context-aware applications; with context being stored as tuples within a local tuple space on each node in the network. Each tuple may specify a context rule which dictates the conditions that must be met by another node before it is able to perceive the tuple. Context rules are defined as a conjunction of tuples and methods (defined in the AmbientTalk scripting language) where any tuples included in a rule can use their field values as variables within methods, allowing for quite sophisticated conditionals. See table 3.1 for an example taken from a "Cops and Robbers" game called Flikken [13]. The example shows a tuple being injected into the space with a context rule stipulating that in order for a device to perceive the tuple (which represents a grenade) it must have a group id of GANGSTER and be geographically proximate to the location field specified in the inserted tuple.

The system also supports a leasing model which removes tuples from remote spaces after a defined period and a propagation mechanism which allows developers to scope the distribution of tuples in the network.

The tuple space supports both public and private tuples. Private tuples remain solely in the local tuple space whereas public tuples may be perceivable by remote nodes (if the specified context rule is satisfied). To support the removal of tuples the originator node injects an anti-tuple into their local space which is identical (apart from an "anti" marker) to the tuple they wish to remove. This anti-tuple will then be propagated through the network, eliminating it's corresponding tuple whenever it is encountered.

3.10 A Context-Aware Middleware Taxonomy

There are several observations that can be made in light of the above discussion.

Firstly, in order for a system to allow easy modification and extension of ontologies it is necessary for them to be de-coupled from the application code. In addition, to allow for flexibility, a modelling technique which supports the definition of relationships between entities must be used. The OWL language would seem to be the most suitable tool for achieving these goals, as illustrated within several of the systems surveyed above.

Finally, systems tend to provide access to their services via a publish/subscribe methodology which has the benefit of conserving bandwidth and power, as well as conceivably reducing application complexity.

Each of the discussed systems is summarised in Table 3.2.

3.11 Summary

In this chapter we have provided definitions of both context and context-awareness and have categorised context-aware systems according to the types of functionality they provide. We then discussed the different aspects of context, both in terms of their categorisation and by providing several concrete examples. Next we looked into means of representing context information (location in particular) and outlined each of the models used by current systems.

We provided several criteria for evaluating existing context-aware frameworks (flexible evaluation, extension, heterogeneous interoperability and centralisation) and then evaluated nine such systems accordingly. We finish with a taxonomy which illustrates that none of the systems surveyed completely satisfy all of the criteria proposed and that this necessitates the development of a new system to fulfil these requirements.

	Flexible Eval	Ontology Extend	Heterogenous Op	Decentralisation
Context Toolkit	Widget Composition (1*)	Tight coupling with sensor code (1*)	HTTP/XML (3*)	Centralised aggregation server (2*)
CoBra	Inference Engine & richer semantics (3*)	OWL (4*)	SOAP/FIPA-ADL/XML (5*)	Central "context broker" (2*)
Context Management Framework	Supports logical connectives AND, OR and NOT (4*)	High level contexts (3*)	RDF/XML (4*)	Centralised "context manager" (2*)
Hydrogen	Limited (1*)	Tight coupling with code (1*)	XML (3*)	Decentralised, but limited (4*)
SOCAM	Sophisticated reasoning support and logical connectives (5*)	OWL (5*)	OWL (4*)	Partially decentralised (3*)
Vehicular Networking	Inference-based service subscription (3*)	OWL (5*)	Java RMI (2*)	Partially decentralised (vehicle-to-vehicle), partially localised (GSM support) (3*)
MetroSense	N\A - Focuses exclusively on decentralised data requests/receipt	N\A - Focuses exclusively on decentralised data requests/receipt	Definition of common interfaces/communication primitives (4*)	Ad-hoc movement based delegation network (5*)
Coalition	SQL-like Domain-attribute queries + constraints (4*)	Arbitrary num of key-value properties (raw sensor or high level) (3*)	SOAP (5*)	Centralised Management service, Decentralised processing (3*)
TOTAM	Conjunctions, not disjunctions, XOR or nested blocks (3*)	Define new methods in AmbientTalk (4*)	AmbientTalk (2*)	Decentralised (5*)

Table 3.2: Summarising the Toolkits with ratings (1* to 5*)

4 Context-Aware Content Distribution

4.1 Introduction

This Chapter discusses our MediateSpace framework in detail. We begin by describing our language, the network topology and network protocols. We then discuss our methodology for evaluating the contextual conditions provided by our language. Finally, we provide a much expanded discussion of the pervasive advertising application outlined in Chapter 1.

4.2 The MediateSpace Language

We based our work on the *Tuple space paradigm* [39], with all system structures and payload data being stored in tuples. We extended the paradigm with notions of context and decentralised communication.

4.2.1 The Contextual Language

The contextual language makes use of predicate logic to make it as flexible as possible. Specifically, it supports universal and existential quantification ($\forall$, $\exists$), conjunctions, disjunctions and negation.

Conditions may be connected together using two types of command. The first simply binds commands together using logical connectives. The second type is much more powerful as it allows you to use quantifiers. In predicate logic the $\forall$ quantifier stipulates that all members of a domain satisfy a given predicate; whereas the $\exists$ quantifier stipulates that at least one member of a domain satisfies a given predicate. In our language it is used to stipulate that all conditions evaluate to true or that at least one of the conditions evaluate to true respectively. We have extended the $\exists$ quantifier with two additional parameters that allow us to stipulate that O conditions are true, where n $\leq$ O $\leq$ m ($\exists$ n .. m). Curly braces are used to separate blocks of conditions so that they may be connected together into increasingly complex statements.

```
@Location.XY(51.06, 0.127, 0.5);@
{
  Std.Compare(Time.getTime(), ">", 08:00:00) &&
  Std.Compare(Time.getTime(), "<", 17:00:00) &&
  {
    Season.Season(SUMMER) || Season.Season(AUTUMN) ||
    Season.Season(SPRING)
  } &&
  Std.Compare(Distance.MaxWalkDist(), ">=", 10) &&
  Std.Compare(Attributes.scenicView(), "==", true)
}
```

(a) Example Contextual Conditions using Conjunctions and Disjunctions

```
@Location.XY(51.06, 0.127, 0.5);@
{
  forall
  Std.Compare(Time.getTime(), ">", 08:00:00),
  Std.Compare(Time.getTime(), "<", 17:00:00),
  Std.Compare(Distance.MaxWalkDist(), ">=", 10),
  Std.Compare(Attributes.scenicView(), "==", true);
} &&
{
  exists (1, 1)
  Season.Season(SUMMER),
  Season.Season(AUTUMN),
  Season.Season(SPRING);
}
```

(b) Example ∀ and ∃ Conditions

Figure 4.1: Representing the same Context using Different Commands

A GPS location may be specified at the head of the condition to ensure that only Participants within the set area can receive the message. The Location Contract takes three parameters: latitude, longitude and a radius (in Km) which surrounds the point specified in the first two parameters.

Figure 4.1 illustrates two ways of representing the same condition. Subfigure 4.1a uses only conjunctions and disjunctions whereas Subfigure 4.1b uses ∀ and ∃ quantifiers. These examples also illustrate the use of blocks and nested blocks. Subfigure 4.1b uses two blocks connected via a conjunction, illustrating how blocks can be used to separate multiple uses of quantifier commands. Subfigure 4.1a uses a nested block to separate the Season Contracts into a different scope, allowing us to specify the different possible values for Season in a concise way.

4.2.2 Defining Context

In order to reason about contexts it is necessary to:

1. Create a model of the context.
2. Map the values from an appropriate sensor onto the model.

The former is achieved by defining a `Context` Tuple while the latter requires definition of a `ConcreteContext` Tuple. Contexts and ConcreteContexts are analogous to object-oriented interfaces and classes respectively. `Contexts` allow the definition of a list of abstract method signatures and an ontology of possible values. ConcreteContexts implement the methods and ontology with regard to a particular sensor.

The system supports six data types (`Boolean`, `String`, `Integer`, `Double`, `Date`, `Time`, `Ontology`). The `Ontology` type accepts any value defined as part of the ontology for the current Context.

Each ConcreteContext must specify a driver which is a small program that interfaces with the appropriate sensor to return values for each of the Contracts defined within the implemented Context Tuple.

Figure 4.2 provides examples of these structures using the Distributed Geocaching application from Section 1.4.2. The SearchParams ConcreteContext implements all of the available Contexts so its driver must provide implementations for all possible Contracts. The driver program for each ConcreteContext is specified as the value of the "contextDriver" key within the Meta block of the ConcreteContext.

4.2.3 Exchanging Context Values

There will be circumstances where a participant cannot access the types of context necessary to receive a message. In these circumstances, participants can request information from another participant via the exchange of `ContextRequest` and `ContextValue` tuples. However, care must be taken because contexts may only be applicable under certain conditions. For example, a Location context may only be valid if the participant is less than 1/4 Km away. The maximum distance for a given Context is specified as the value of the "maxSpatialDistance" key within the Meta block of the Context Tuple.

In some cases context must be derived locally to be valid. Examples include age, gender, budget and personal interests. These details can be data mined from Facebook profiles or E-Mail; or alternatively they can be explicitly specified by the user. See the "ExplicitPersonalInfo" and "SearchParams" ConcreteContexts in Figures 4.11 and 1.3 respectively for examples. This locality property can be enforced on applicable Contexts by setting the "maxSpatialDistance" key to zero.

Each regional node is aware of its position within geographical space and the positions of the regional nodes to which it is bound. These properties make it straightforward to restrict access by distance. Regional nodes may continue issuing requests iteratively until all further regions lie beyond the maximum distance from the original requesting node.

```
Context Season {
  Meta {
    ((tupleName, "P3−Season−e47e3a884ce5d6a411d8"),
     (maxSpatialDistance, 50.0),
     (originatorId, "P3"), (sourceId, "P3"));
  }
  Contracts {
    bool  Season(ontval ont);
  }
  Ontology {
    SUMMER, WINTER, AUTUMN, SPRING;
  }
}

ConcreteContext SearchParams implements Time, Season,
                                        Distance, Attributes {
  Meta {
    ((tupleName, "P1−SearchParams−0ab1b619d264c83a420a"),
     (contextDriver, "uk.co.dmatthews.mediateSpace.SearchParams"),
     (originatorId, "P1"), (sourceId, "P1"));
  }
}

ConcreteContext DeviceCalendar implements Season {
  Meta {
    ((tupleName, "P3−DeviceCalendar−090515a65ce7c2b68da9"),
     (contextDriver, "uk.co.dmatthews.mediateSpace.DeviceCalendar"),
     (originatorId, "P3"), (sourceId, "P3"));
  }
}
```

Figure 4.2: Contexts and ConcreteContexts from the Geocaching Application

4.2.4 Exchanging Messages

`Message` Tuples contain the information which is distributed across the network. It is composed of a contextual condition, an *Advert* (used by participants to decide whether a message is applicable) and one or more payload modules which contain the data of the tuple. Figure 4.3 illustrates an example Message Tuple from our Geocaching application. In addition to a contextual condition the Message specifies several modules of information with corresponding adverts. The user will receive the Advert block of the Message before the content. This allows them to specify exactly which blocks of information they want to receive. For example, a user may wish to only receive the Meta and Logs blocks because they are on a cellular network and do not want to waste bandwidth on the photographs.

Users obtain Messages by issuing `MessageRequest` Tuples into the network, containing a contextual condition known as the guideline condition. This is used to filter out any Messages which do not satisfy the condition.

The remaining messages are candidates for delivery.

The guideline condition is defined as the union between a condition specified by the user and any context information which has already been evaluated and is available locally.

The process of querying the system for messages is undertaken in several steps and involves the `MessageMatch` tuple transitioning sequentially through three states. The MessageMatch tuple acts as a container for various structures exchanged between the requesting and requestee nodes. For instance, it is used to contain the adverts for any candidate messages and will be used to deliver the filtered versions of these Messages when appropriate.

The protocol for Message lookup is explained is much greater detail in Section 4.3.5.

4.2.5 Network Tuples

There are three types of tuple which are used purely to perform network communication tasks. The `Bind` and `Unbind` Tuples are used when binding to and unbinding from an external node respectively. The `Signal` Tuple is used to represent events. For example, a Signal Tuple is issued to an external node to inform it that all ConcreteContext Tuples have been dispatched to it.

In addition to containing source and destination information, the Bind Tuple also contains the geographical location of the binding node and a field indicating the status of the Bind operation. For example, the status field may contain a REQUEST, REPLY or REJECT value. Bind requests are rejected when the requestee is already bound to a defined maximum number of neighbours.

Table 4.1 provides a summary description of each of the Tuples discussed in the preceding section.

4.3 The MediateSpace Network

This section discusses the details of the MediateSpace network. Specifically, we discuss the abstractions used for network communication, the different types of supported node, the network topology and the protocols used during communication.

4.3.1 Overview

All nodes have access to two tuple spaces, known as the internal and external tuple spaces. The exact semantics of each space differ depending on the type of node but in all cases the tuple spaces provide an abstraction which allow nodes to communicate with one another. That is, the node places an addressed tuple into one of their spaces and it will be delivered over the network to the

```
MessageTuple {
  Meta {
    ((tupleName, "cache-b8fa8a79847db01aa328"), (msgUniqueId, 865),
     (originatorId, "P2"), (sourceId, "P2"), (destinationId, "P5"));
  }
  Condition {
    @Location.XY(51.06, 0.127, 0.5);@
    {
      Std.Compare(Time.getTime(), ">", 08:00:00) &&
      Std.Compare(Time.getTime(), "<", 17:00:00) &&
      {
        Season.Season(SUMMER) || Season.Season(AUTUMN) ||
        Season.Season(SPRING)
      } &&
      Std.Compare(Distance.MaxWalkDist(), ">=", 10) &&
      Std.Compare(Attributes.scenicView(), "==", true)
    }

    Advert {
      (Meta, ["Basic Info",
              "Title, desc, hint, difficulty, terrain, size"]),
      (Logs, ["Logs", "Log info for this cache"]),
      (Photos, ["Photos", "Photos of the cache"]);
    }

    PayLoadModules {
      Meta { ((title, "University of Sussex"),
              (description, "The cache is on the boundary walk"),
              (hint, "in a tree"),
              (difficulty, 2), (terrain, 2), (size, 2));
      }
      Logs { [ ((title, "Found it!"), (date, 20/08/2014),
                (body, "Solved. quick trip to find it. TFTC")),
               ((nextLink, "cache-444b7c605dd7e22fdac6"));]
      }
      Photos { [ ((photo1, "d76357f331433b1eec89c35e82b"),
                  (nextLink, "cache-1ea90295d182746de156"));]
      }
    }
  }
}
```

Figure 4.3: An Example Geocache Message

Tuple Type	Description
Context	A model for a type of contextual information. This involves the definition of one or more Contracts and optionally an ontology
ConcreteContext	An implementation of one or more Contexts; providing a driver program which interfaces with an appropriate sensor and returns results
ContextRequest	For requesting contextual information from nearby Participants. Context information generally only remains valid over have a certain distance so requests should not be made to nodes that exceed this distance
ContextValue	Represents contextual values. ContextValue tuples only remain valid for a finite period of time
Message	Represents the application data. Each Message has a condition which must be satisfied and a number of payload modules containing the actual data. Adverts can also be specified to allow the user to choose which of the modules to receive
MessageRequest	Used to query the network for Messages. They contain a guideline condition defined as the union of a condition specified by the user and anylocally available context information
MessageMatch	Used as a container for delivering data between the requesting and requestee nodes. Its contents may include message adverts and the actual messages themselves
Bind	Used to issue Bind messages between two nodes. Bind Tuples carry the location of the dispatching node and can have one of several states (REQUEST, REPLY, REJECT)
Unbind	Used to issue an Unbind request to a node
Signal	Used to indicate the occurance of an event to an external node (e.g. end of data)

Table 4.1: A Summary of all MediateSpace Tuples

appropriate node. Each Tuple Space also has a number of subspaces which are used to store different types of tuple.

The MediateSpace language structures are parsed into Tuples with named fields being used to store the associated data. For example, each tuple will have a field containing its name with the key TUPLE_NAME.

Our system supports two types of node:

- Regional
- Participant

We now discuss each of these in turn.

4.3.2 Regional Nodes

The Regional node is responsible for most of the computationally expensive operations and network communication. Specifically, they handle the evaluation of contextual conditions and the majority of network communication necessary to lookup Context Values and Messages. It is intended that Regional nodes are run on the more capable machines of the network with fast processors and significant amounts of RAM.

As discussed previously, each node has an internal and an external tuple space. Regional nodes use both spaces to communicate with remote nodes to which they are bound. The internal space is used to communicate with bound Participant nodes and the external space is used to communicate with bound Regional nodes. Both tuple spaces support five subspaces, allowing the Regional node to read and insert tuples of all types during communication with remote Participant and Regional nodes. These subspaces are illustrated in Figure 4.4.

4.3.3 Participant Nodes

In general Participant nodes will run on less capable devices such as phones and tablets. These nodes can request and provide contextual information and can issue requests for Messages from remote nodes. We anticipate that Participant nodes can run adequately on devices with lower grade processors and comparatively small amounts of RAM because they are not responsible for any complicated or memory intensive operations. The main responsibilities of Participants are to parse and dispatch tuples over the network and to interface with onboard sensors to obtain context information for local use or for use by another geographically proximate Participant.

The internal space of Participant nodes is not available to the network and it's contents are loaded from disk at startup. The space is used to store Context, ConcreteContext and Message tuples. The external space is used to communicate with the Regional node to which it is bound. The external space supports all of the subspaces in Figure 4.4. The internal space is only required

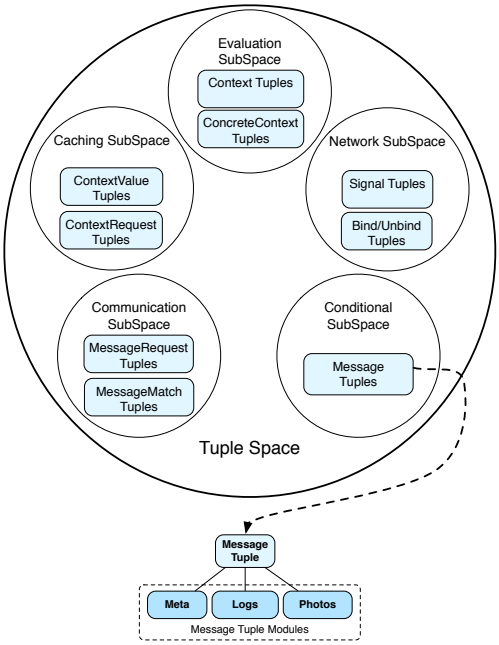


Figure 4.4: The Five Supported Subspaces

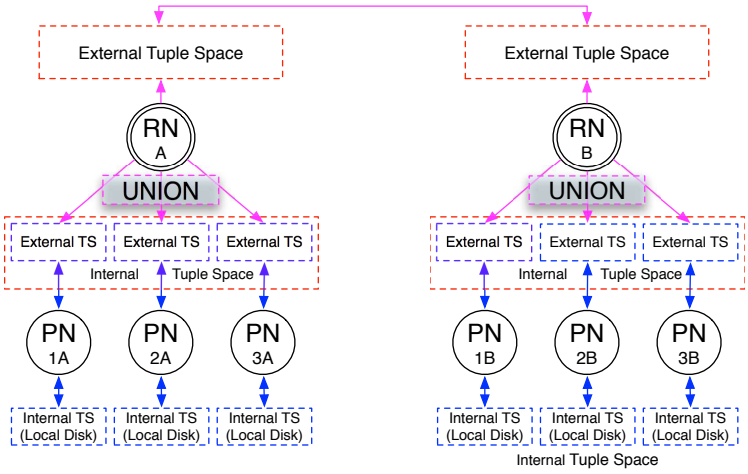


Figure 4.5: The Tuple Space Network Abstraction

to handle Contexts, ConcreteContexts and Messages so only the Evaluation and Conditional subspaces are activated.

As discussed, nodes can communicate with one another via the tuple space abstraction. The Regional internal space can be viewed conceptually as the union of the external spaces of all bound Participants. The internal and external spaces of both types of node are illustrated in Figure 4.5.

4.3.4 Network Topology

The topology of the network has the following basic properties:

1. Participant nodes attempt to bind to the most geographically proximate Regional node in the network
2. Regional nodes attempt to bind to the n most geographically proximate Regional nodes in the network

Both types of node periodically rebind to ensure that they always remain bound to nearby nodes.

4.3.4.1 The Location Spatial Index

Spatial indexes such as the R-Tree are data structures that allow the quick insertion and lookup of multi-dimensional data and also provide a facility for locating the N-nearest indexes to a particular point. Nodes are able to efficiently lookup their closest Regional nodes because each Regional node registers itself with a distributed spatial index at startup, and periodically after that if the node is non-static. Regional node locations are stored as two-dimensional indexes consisting of latitude and longitude.

4.3.4.2 The Message Spatial Index

Messages are also stored in a distributed spatial index, with the number of dimensions used equal to the number of Contracts in all Context Tuples. Each Contract corresponds to a single dimension of the index and each Message is indexed according to it's contextual condition. Contracts can represent single points on a dimension or a range of values. For example, the Contract:

```
Std.Compare(Time.getTime(), "==", 08:00:00)
```

would be represented as a single point whereas the Contract:

```
Std.Compare(Time.getTime(), ">", 08:00:00)
```

would represent the range of values from 8 AM to 11:59:59 PM. See Figure 4.6 for an example of an R-Tree representing an application with three Contracts. The point data represent conditions where a single value has been specified for each Contract, whereas the internal rectangles represent conditions where all Contracts have specified ranges. Conditions can also specify a combination of point and range data. The mapping of conditions to indexes is discussed in detail in Chapter 5.

When a Message Request is issued Regional nodes map the guideline condition specified within it to an index. This index is then used to perform an intersection-based search on the distributed data structure, resulting in a number of candidate messages that should be explored further.

The location index is available to Regional and Participant nodes as both are required to lookup nearby nodes to bind to. The Message index is available

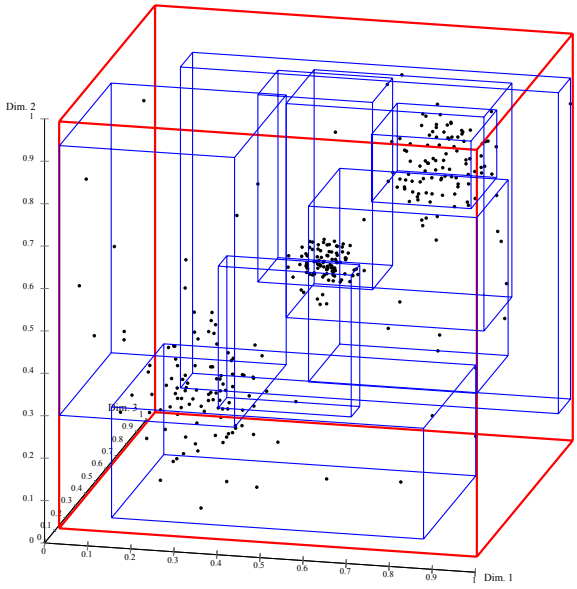


Figure 4.6: A Three-Dimensional R-Tree[21]

only to the Regional nodes as they are responsible for all Message lookup and evaluation.

4.3.5 Network Protocols

We have defined a number of communication protocols that implement the semantics given above. This section discusses each of the following protocols:

1. Participant and Regional Binding
2. Participant and Regional Unbinding
3. Context Requests
4. Message Requests
5. Message Distribution

4.3.5.1 Bind Protocol

We shall first discuss the bind protocol for Participant nodes and then for Regional nodes.

In order to request messages and share context information Participant nodes must first bind to a geographically proximate Regional node. The node first obtains the address of the geographically closest Regional node and dispatches a bind request. The node includes its location within the request so that the Regional node can make a record of it if binding is successful. Location is stored during binding so that the Regional node can determine whether a Context Request can be validly dispatched to the Participant without ex-

ceeding the maximum distance specified for the Context. If the Regional node is not already oversubscribed it will reply to the Participant, accepting the request.

The Participant will then dispatch copies of all their local ConcreteContext tuples to the Regional node, which informs the node of the types of Context the Participant supports so that Context Requests can be dispatched to it when appropriate. The Participant dispatches a Signal tuple to indicate that all available ConcreteContexts have been sent and the Regional node acknowledges this with its own Signal. This signifies the end of the initialisation phase of binding and the Participant is now ready to receive Context Requests.

If the Participant has any Message or Message Request tuples stored locally they are now shared with the Regional node. Upon receipt of Message tuples the Regional node will distribute them to the most appropriate Regional node in the network. The message distribution protocol is explained in Section 4.3.5.5.

If the Regional node rejects a bind request the Participant will attempt to bind to the next closest node, and so on until binding is successful or a defined maximum number of attempts are exhausted. A maximum number of attempts is specified to safeguard against the possibility that the Participant has many failed attempts and eventually binds to a node a great distance away. The maximum number of attempts should depend on the types of Context supported by an application. For example, in our Geocaching application the maximum valid distances for the Time and Season contexts would likely be very large as their values tend to only change when country or continent boundaries are passed. In the Pervasive Advertising application the number of attempts should probably be much lower as location information should be as accurate as possible.

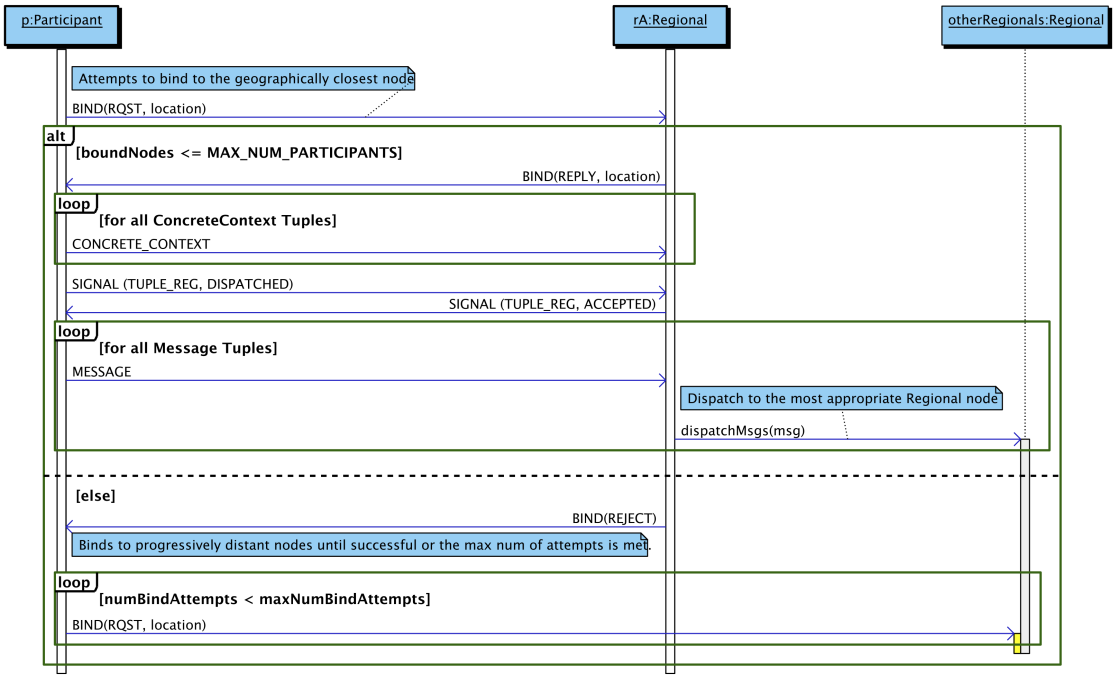
The bind protocol for Regional nodes is similar to the Participant protocol. The only differences are that there is no initialisation phase and each Regional node attempts to bind to their N closest Regional nodes rather than just the closest.

Each Regional node maintains separate BOUND and BOUND_TO lists, meaning that binds do not need to be symmetric. That is, node A can be bound to node B without node B also being bound to node A.

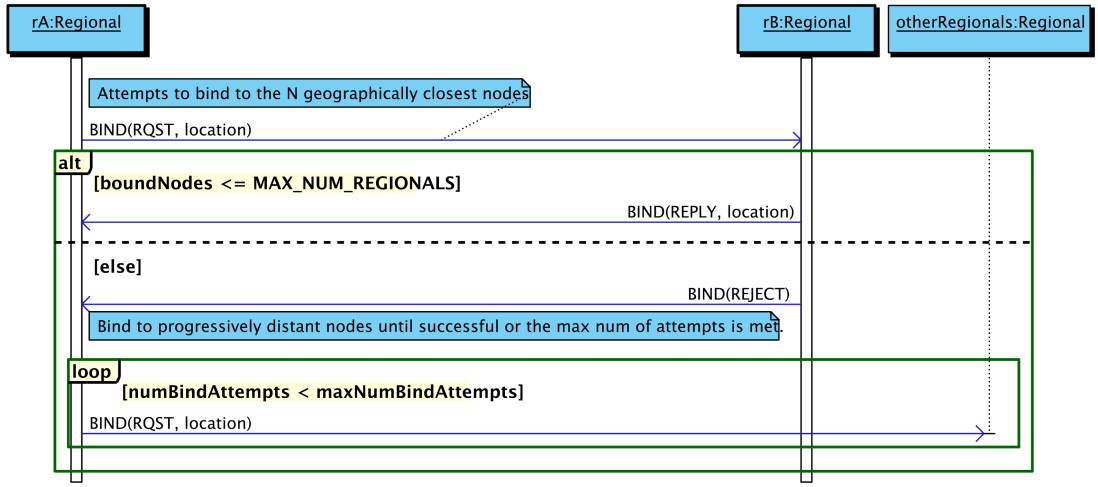
The bind operations are summarised in Figure 4.7.

4.3.5.2 Unbind Protocol

The Unbind protocol is straightforward and simply requires that an unbind tuple is dispatched to the appropriate node. The receiving node will remove the requesting node from its bound and location lists.



(a) The Participant Node Bind Protocol



(b) The Regional Node Bind Protocol

Figure 4.7: The Bind Protocols

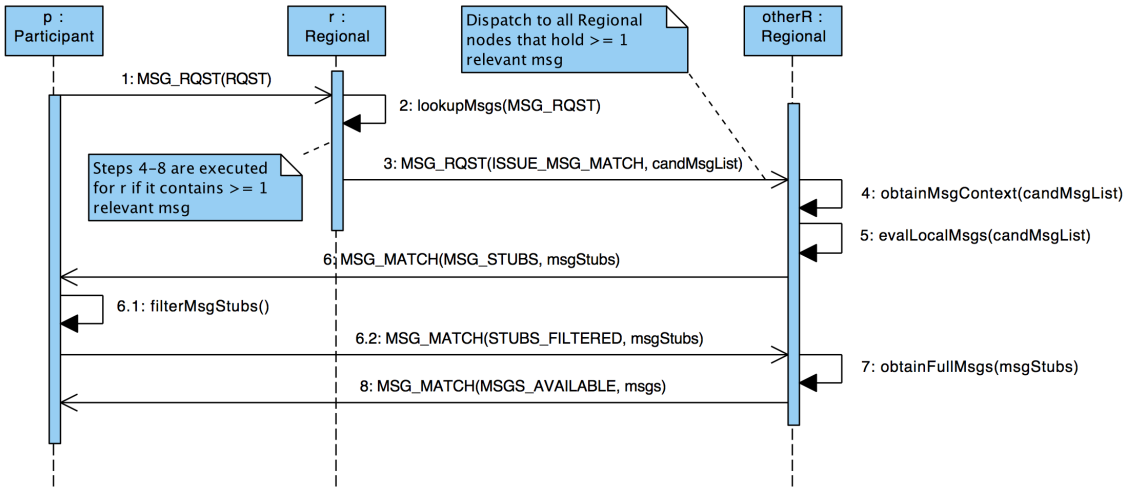


Figure 4.8: The Message Request Protocol

4.3.5.3 Message Request Protocol

The message request protocol is instigated when a Participant dispatches a MessageRequest tuple to their bound Regional node. When the MessageRequest tuple is received the contextual condition defined within it is parsed into one or more spatial indexes. These indexes are used to query the Message tree which returns a list of reference objects. Each object contains the address of a Regional node and a list of candidate Ids which can be mapped to messages within the node. The MessageRequest tuple is then dispatched to all nodes for which a reference object was found. Each Regional node (including the local node) will then begin the process of evaluating each candidate message.

Before each candidate message is evaluated the node will determine the Context Values needed to perform said evaluation. Requests are then dispatched for these values (discussed in Section 4.3.5.4). The protocol waits for a short period before continuing in order to allow any requested Context Values to arrive. Each message is evaluated using the process in Section 4.3.6. Any succeeding messages have their Advert section extracted and dispatched to the Participant node for perusal within a MessageMatch tuple. The Participant filters out any modules they do not wish to receive and dispatches the tuple back to its originating node. Finally, the Regional node collates the modules requested and dispatches them to the Participant. The protocol is summarised in Figure 4.8.

4.3.5.4 Context Request Protocol

As discussed in the previous section the Regional node will issue Context Requests for any required context information. Before issuing requests to

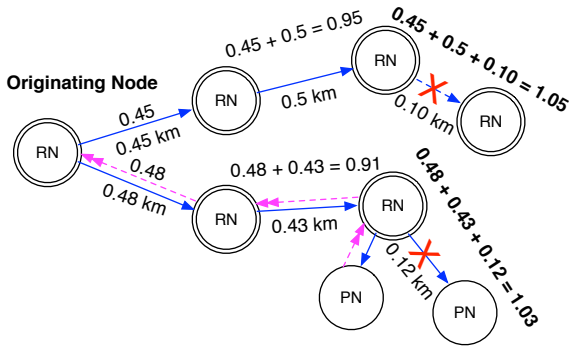


Figure 4.9: Context Requests and Request Buffering (max dist: 1 km)

external nodes the node will check whether the context is available in the internal space. Context Value tuples will be available for a given context if a previous request for them was made and answered and they have not expired.

If the context cannot be located locally the node determines whether any of their bound Participant nodes would be able to provide it. As context information tends to remain valid only within a finite geographical distance we need to ensure that no requests are made to nodes that exceed this distance. As noted earlier location information is stored locally for each node during the binding process. The maximum distance is obtained from a field in the appropriate Context tuple and the distance between the local node and a prospective Participant node is calculated. The ContextRequest tuple is only issued if the Participant is within the maximum distance.

If the context information is not available from a Participant, requests are made to the bound Regional nodes. The same process is followed as above to ensure that the nodes are within a valid distance.

Whenever a Regional node receives a Context Request from another Regional node it saves the request in memory for a finite period so that if the requested Context Value is later inserted into the local space it can be returned to the requesting node. This process is illustrated in Figure 4.9 where the single headed solid arrows represent Context Requests and the double headed dashed arrows represent the return journey of the Context Value tuple.

The protocol is summarised in Figure 4.10.

4.3.5.5 Message Distribution Protocol

We attempt to store Message tuples in a Regional node which is geographically close to where most requests for the Message originate. When each Message is received by a Regional node it obtains the location specified at the head of the contextual condition for the Message. It then calculates the distance between these coordinates and the N closest Regional nodes (including the local node). We multiply these distances against the total number of Messages

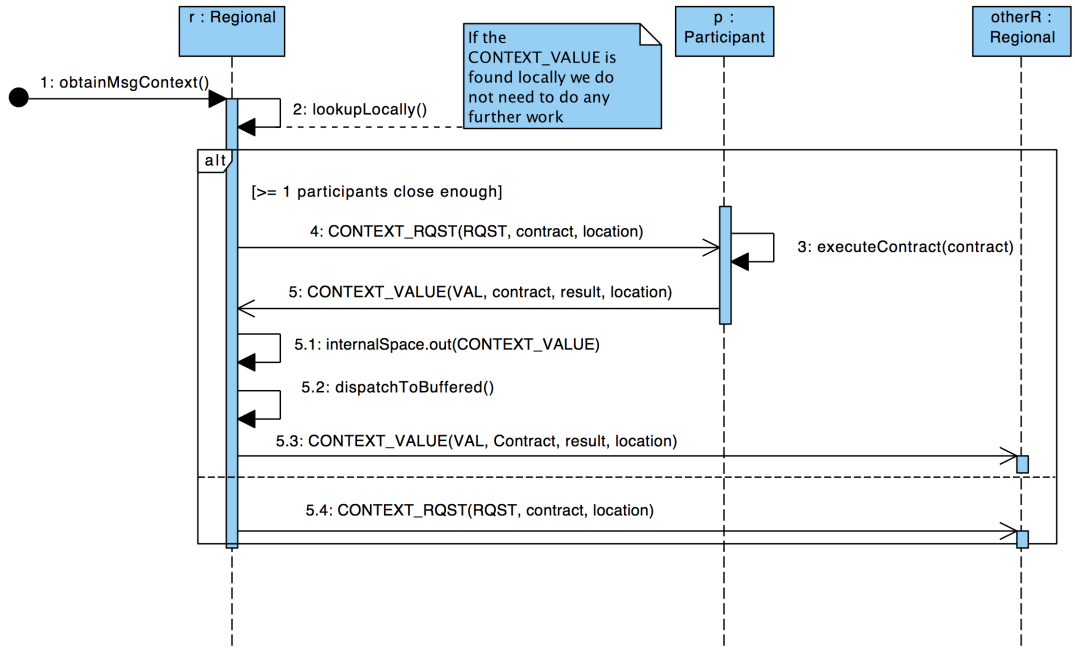


Figure 4.10: The Context Request Protocol

stored on each node to support a simple form of load balancing. Thus, our message distribution protocol attempts to appropriately store Messages based on location and load.

4.3.6 Evaluating Contextual Conditions

Our middleware allows the sharing of context and messages. In order to achieve context sharing the Regional nodes need to be able to establish which of their bound Participants can supply the necessary context values. To support message exchange the Regional nodes need to be able to evaluate the contextual conditions specified with each message.

To establish Participant context support each Context and ConcreteContext is translated into the web ontology language (OWL[1]) in a form that retains the relationships between them. For example, if ConcreteContextA implements ContextA a subclass relationship between these structures will be created in OWL. An OWL reasoner can then be used to construct a list of ConcreteContexts supporting a given Context.

Contextual conditions are evaluated by translating the conditions into OWL. We also translate any available ContextValue tuples so that the reasoner has a complete record of available context during evaluation.

[1]http://www.w3.org/TR/owl-features/

The OWL representations for Contexts, ConcreteContexts, ContextValues and all types of contextual condition are described in Chapter 6.

The following subsection provides a much expanded discussion of the pervasive advertising application outlined in Chapter 1.

4.4 Pervasive Advertising

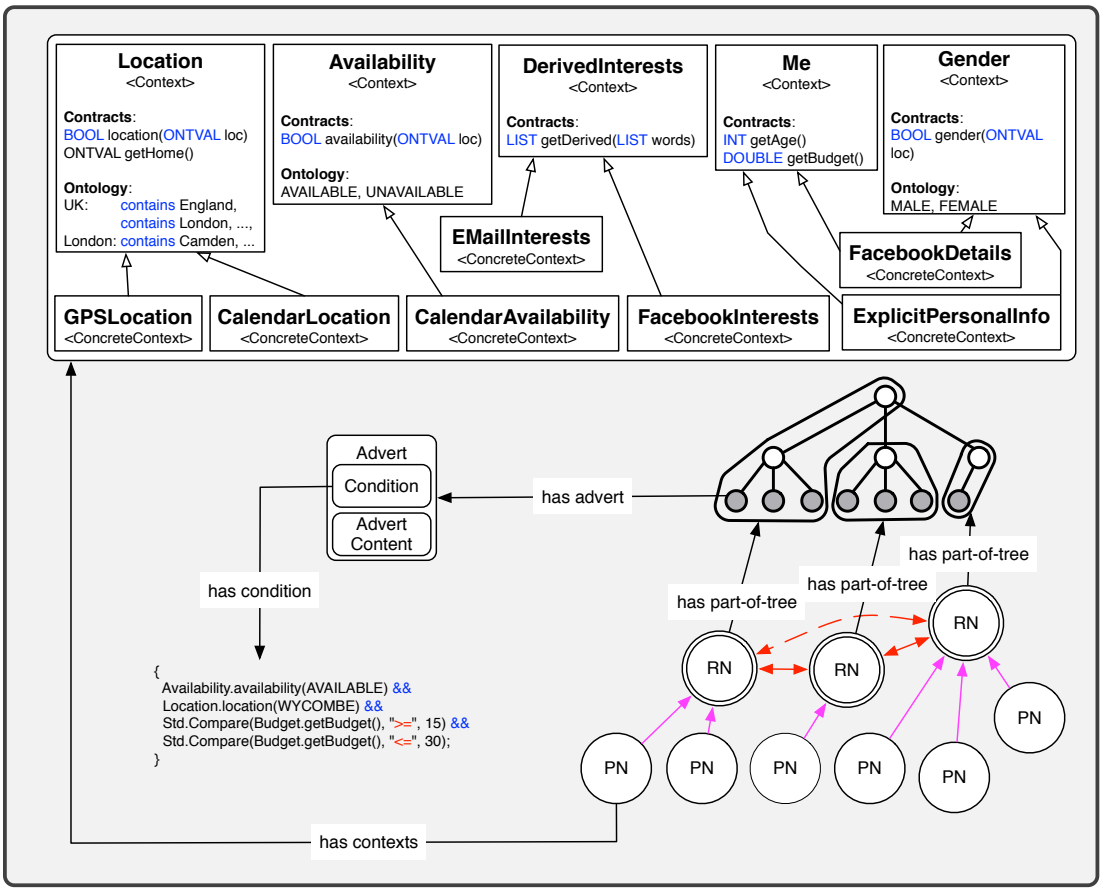
The Pervasive Advertising application discussed in this subsection is based on our 2012 paper on the same subject [60]. This paper was written in conjunction with my supervisors Dan Chalmers and Ian Wakeman and was influenced by Chalmers et al.'s paper on comparing context relationships [16]. However, the sections reproduced here are the candidates own work.

Existing pervasive advertising frameworks such as MyAds [29] and MobiAd [47] focus primarily on matching adverts based on explicitly entered information (such as user demographics and interests), derived information (e.g. through the parsing of browser history, Facebook or E-Mail) and location. A number of commercial mobile services also have this focus. The AdMob[2] service allows developers to insert adverts into their mobile applications and also allows mobile-specific enhancements to be made to Google results (e.g. clickable telephone numbers, maps and distance-from information). iAd[3] allows developers to insert adverts into their iOS applications and includes contexts such as music and network availability (e.g. WiFi or 3G). We aim to demonstrate how these types of context can be represented using the MediateSpace middleware.

Our example of use considers a shopping scenario where stores wish to advertise their products to appropriate customers. They wish to target individuals based on their budget, their proximity to the store, their availability and their shopping interests. Our system allows stores to distribute adverts with contextual conditions attached; delivering the adverts to only those individuals whose context matches. Figure 4.11 illustrates a potential design for this scenario with five Contexts and seven ConcreteContexts.

Customers will obviously want the adverts they receive to be as relevant as possible. Hence, our advertising platform should rank each matching advert by relevance before delivery. Our ranking algorithm is based on Google Adwords[4]. We will now briefly discuss the Adwords model and then move on to discuss our Context-Aware Bidding model.

[2] http://www.admob.com/
[3] http://advertising.apple.com/uk/
[4] http://adwords.google.com/



Context	Description
Location	Provides an ontology which represents different regions of the United Kingdom. **location(ONTVAL)**: Accepts an ontology value as parameter and returns a boolean indicating whether the user is within the specified location. **getHome()**: Returns the ontology value that represents the user's place of residence. **ConcreteContexts**: Using GPS or the users calendar.
Availability	**availability(ONTVAL)**: Accepts an ontology value as parameter and returns a boolean indicating whether the user is currently available. **ConcreteContexts**: Via the users calendar.
DerivedInterests	**getDerived(LIST)**: Accepts a list of words takenfrom a data source and outputs a list of derived words. **ConcreteContexts**: Data mining the users E-Mail or Facebook account.
Me	**getAge()**: A single Contract for obtaining the age of the user. **getBudget()**: Returns a floating-point value representing the budget of the user. **ConcreteContexts**: Explicit information input by the user or via Facebook.
Gender	**gender(ONTVAL)**: Accepts an ontology value as parameter and returns a boolean indicating whether the user is of a specified gender. **ConcreteContexts**: Explicit information input by the user or via Facebook.

Figure 4.11: A Summary of our Pervasive Advertising Application

4.4.1 Google Adwords

We focus our discussion on Google AdWords because it holds the vast majority of market share (81% [70]) and is largely representative of most other major networks. Google AdWords allows businesses or individuals to advertise their products within Google search results and on participating websites. The AdWords user creates campaigns and specifies keywords to be matched, a daily budget and a maximum cost-per-click for each chosen keyword. The user is usually charged less per-click than their specified maximum (discussed later). AdWords supports a myriad of customer targeting options and uses a sophisticated advert ranking mechanism. We now discuss these.

Advertisement Targeting In order to target likely customers an AdWords user supplies a number of keywords for each advert they wish to display. Adverts are shown when these keywords match Google search terms or the theme of a participating website. Additional parameters can be specified to further improve the likelihood of adverts being seen by interested parties. For instance, display may be restricted to a particular geographical region (e.g. city, country or radius around a point), by language, by device type or even by operating system and network carrier in the case of mobile devices. When embedding an advert in a website it is also possible to restrict by visitor demographic (gender and age).

Advertisement Ranking The page number and position (AdRank) an advert achieves within the results depends on both the maximum cost-per-click attributed to the keyword and the quality of the advert. Quality scores are within the range [1, 10] and are based on a number of metrics aimed at determining how relevant an advert is to a user's query. These include click-through-rate (CTR) and the relevance of keywords to the advert text. The actual cost-per-click (CPC) charged depends on the user's maximum CPC, their quality score and the AdRank of their closest competitor. The Formulae for calculating AdRank are defined below.

$$AdRank = MaxCPC \times QualityScore$$

$$ActualCPC = \frac{NextHighestCompetitorsAdRank}{YourQualityScore} + \$0.01$$

4.4.2 Context-Aware Bidding

In our context-aware bidding model advertisers bid for the right to display their adverts to users within a specified context. The system generates a quality score for each contextual condition which is then combined with advertiser bids to calculate the actual cost-per-click paid by the advertiser. This is based on the AdWords formula specified in section 4.4.1.

Quality score is based on *specificity*; the more specific a condition, the higher the quality score it will attain. For example, a condition targeting the whole of the United Kingdom would likely (depending on the specificity of the other contexts used) receive a lower quality score than a condition targeting a single city within the UK.

Our system allows context to be specified using three types of representation:

Value ranges Ranges of values which the user's context may coincide with. For example, if an advertiser wished to only advertise to users with a budget between 50 and 100 pounds (£), they could specify this using the value range *Budget(50 - 100)*.

Hierarchical ontologies Ontologies represent concepts and the relationships which exist between them. Hierarchical ontologies are useful for representing many domains such as geographical location. This could be modelled using containment relationships between the concepts; so, for example the European Union contains the UK, and the UK contains London.

Keywords Keywords represent the theme of the advert. For example, an advert for a company developing software metric visualisation software may specify the keywords *Keywords(metrics, visualisation, software)*.

The value range and hierarchical ontology representations are based on the relationships discussed in [16].

To calculate the quality score we first have to calculate a score for each aspect of context (*Aspect Score*) within the condition; with each score falling within the range [0, 10]. The quality score is then calculated as the arithmetic mean of all aspect scores with a weight applied to favour those conditions with a higher numbers of aspects. This is defined in Formula 4.1.

Specificity of Value Ranges To calculate the *specificity* of a value range we first calculate an *expected range* ([EA, EB]), which is the estimated maximum range of values that a context will fall into. This is necessary to ensure fair scores as the *possible range* of values may not fit well with the real world. For example, although theoretically temperature could range from [-128, 136] ° F, this is extremely unlikely in practise.

Once the expected range has been calculated, we determine the score for the aspect by calculating the percentage of the expected range occupied by the *Context Range* ([CA, CB]). Note that we take the maximum of (EB, CB) and the minimum of (EA, CA) to ensure that we take into account situations where the *context range* is wider than the *expected range*. This is defined in Formula 4.2.

Specificity of Hierarchical Ontologies To calculate a *specificity* score for hierarchical ontologies, we assume that concepts get increasingly specific

$$QualityScore = \overline{AspectScores} \times \frac{numAspects}{maxNumAspects} \quad (4.1)$$

$$ValueAspectScore = (1 - \frac{CB - CA}{Max(EB, CB) - Min(EA, CA)}) * 10 \quad (4.2)$$

$$DepthBias = (1 - \frac{BiasValue}{MaxDepth}) \quad (4.3)$$

$$OntAspectScore = \frac{DepthBias}{Ceil((MaxDepth - ActualDepth)/2)} * 10 \quad (4.4)$$

$$KeywordAspectScore = \frac{\overline{KeyWordValues}}{MaxKeywordValue} * 10 \quad (4.5)$$

Figure 4.12: Context-Aware Bidding Formulae

the further down the tree they are. With this in mind, we calculate the aspect score as the difference between the maximum depth and the actual depth (where ontology-depth is zero-based).

We also apply a *Depth Bias* to compensate for the bias resulting from comparing two ontologies of different maximum depth. The effect of the depth bias decreases as the number of levels in the ontology increases. This is based on the intuition that less granular ontologies will not allow the degree of specificity afforded by their more granular counterparts. Depth bias can be any real number in the range [0, 1]. The formulae for these calculations are defined in Formula 4.3 and 4.4.

Specificity of Keywords To calculate an aspect score for keywords, we take into account the importance of each keyword to the individual user. This may be derived in many ways ranging from explicit specification to data mining (web history, Facebook, E-Mail etc).

As multiple keywords may be specified within an aspect, we calculate the mean value from each of the keywords then calculate the relative importance of these terms by dividing against the value of the keyword with highest importance. Note that Keywords are unused during the advert lookup process as their possible values are too diverse and without meaningful semantics. The set of candidate adverts are selected according to their *Value Range* and *Ontology* conditions and keywords are then applied to further refine selection as part of the quality score calculation. This is defined in Formula 4.5.

Note that the calculation of ontology depths and keyword scores is not supported by MediateSpace; this functionality must be implemented as part of the Pervasive Advertising application.

Supporting User Privacy We support user privacy by obfuscating their contextual information, achieved by "widening" the user's context whenever it is requested. For example, a user may have the context *Budget(25)* and

Location(Wycombe). Their privacy settings obfuscate value ranges by ten, and ontologies by one level; meaning that the context delivered is actually *Budget(15 - 35)* and *Location(Buckinghamshire)* (Buckinghamshire contains Wycombe). Obfuscation settings may also be defined explicitly for each aspect of context.

4.4.3 An Example

We present an example of our system in Table 4.2. The relationship between Quality Score and Max CPC can be clearly seen in Table 4.2a. The advertiser at Rank 1 is able to spend 45% less than its closest competitor because of it's highly specific condition. In contrast, the advertiser in Rank 2 was able to push ahead of the condition in Rank 3 despite having a much less specific context by paying 43% more. Also, Table 4.2c illustrates the ontology depth-bias effect; although the AVAILABLE concept is at maximum depth, it's score is heavily penalised because of the ontology's shallowness.

The data used in this example are from the following sources:

Budget Data From the UK 2010 Annual Survey of Hours and Earnings[91]. 320 individuals. Budget is calculated as 10% of the mean weekly salary for their occupation.

Location Data From the Mid-2010 UK Local Authority population records[74]. We created a hierarchical ontology using the *contains* relationship which has 493 concepts and a depth of 7.

Availability Illustrates how the hierarchical ontology formula handles ontologies of significantly different depths. Contains two concepts (AVAILABLE, UNAVAILABLE) and has a depth of 1.

Keywords Data Calculated by counting the number of occurrences of each word within a number of personal documents. Semantically non-relevant words such as "the" were removed and Porter Stemming [69] was used. We identified the top 47 words with a total number of 4596 occurrences.

4.5 Summary

The preceding chapter discussed our MediateSpace middleware. We discussed this in terms of the context-aware language, network topology and protocols and the methodology used to evaluate contextual conditions.

In the following chapter we discuss our algorithm for mapping contextual conditions to a set of one or more spatial indexes.

Advertiser Condition	Quality Score	AdRank	Max CPC	Actual CPC	Rank
Budget (0 - 15) && Location (Wycombe) && Available (AVAILABLE) && Keywords(metrics, visualisation, software)	7.23	14.46	2.00	1.37	1
Budget (0 - 80) && Location (London) && Available (AVAILABLE)	3.96	9.9	2.50	2.48	2
Budget (0 - 15) && Location (Wycombe) && Available (AVAILABLE) && Keywords(tuple, space, java)	6.75	9.79	1.45	1.41	3
Budget (15 - 30) && Location (London) && Available (AVAILABLE)	4.73	9.46	2.00	1.25	4
Budget (15 - 30) && Location (Wycombe) && Available (AVAILABLE)	5.89	5.89	1.00	0.70	5
Budget (0 - 80) && Location (Wycombe) && Available (AVAILABLE)	5.12	4.1	0.80	0.48	6
Budget (0 - 15) && Location (London) && Keywords(metrics, visualisation, software)	4.82	2.41	0.50	0.50	7

(a) Example Conditions

Aspect	CA	CB	EA	EB	Score
Budget	0	15	6.63	209.52	9.28
Budget	15	30	6.63	209.52	9.26
Budget	0	80	6.63	209.52	6.18

(b) Value Range Aspect Scores

Aspect	Concept	Max Depth	Actual Depth	Score
Location	London	7	4	4.64
Location	Wycombe	7	6	9.29
Available	AVAILABLE	1	0	5

(c) Ontology Aspect Scores

Keywords	Max Interest	Score
{metrics, visualisation, software}	285	5.35
{tuple, space, java}	285	3.42

(d) Keyword Aspect Scores

Word	Num.	Word	Num.	Word	Num.
class	285	memory	191	method	134
metrics	115	system	231	tuple	185
name	120	nes	114	table	219
code	147	cpu	118	space	107
visualisation	199	software	143	register	116

(e) Top 15 Keyword Occurrences

Table 4.2: Example Use Case

5 A Context-Based Spatial Lookup Algorithm

5.1 Introduction

In order to support efficient message lookup, it is necessary to index the data. In the case of messages, we search using the contextual conditions discussed in Chapter 4. Thus, it is appropriate to use these conditions as the index for lookup. See Figure 5.1a for an example of a simple contextual condition.

Conditions can make use of an arbitrary number of contracts, and each of these contracts will need to be represented within the index. Thus, this necessitates the use of an indexing scheme which supports multiple dimensions. For example, Figure 5.1a shows a simple condition formed of three contracts from three separate contexts and expresses that the user should be in Brighton in the evening and that the temperature should be between 18.0 and 25.0 degrees. The Time contract will be expressed along one dimension, Location along another and Temperature along yet another. Additionally, as shown in our example, contracts can be defined which express a range of possible values. In our example we can see that the Temperature should be between 18.0 and 25.0 degrees.

Due to the multidimensional and ranged nature of conditions, a spatial index has been chosen and each contract will be represented along a single dimension of the index. We chose to use the R-Tree as spatial index in our implementation but any spatial structure which supports N-dimensional data expressed using min-max notation could be used (e.g. X-Tree).

As explained in Chapter 4, the generated indexes will be used for two purposes:

1. For representing the contextual conditions of messages in our MediateSpace system and storing said indexes in an R-Tree for lookup.
2. For representing the conditions of MessageRequests, whose indexes are used to restrict the search space when looking up messages in the R-Tree.

```
{ Time.Time(EVENING) &&
   Location.Location(BRIGHTON) &&
   Std.Compare(Temperature.temperatureValue(), ">=", 18) &&
   Std.Compare(Temperature.temperatureValue(), "<=", 25); }
```

(a) A Simple Condition

```
Context Time {
  Meta {
    ((originatorId, "drm24"), (appID, "test"));
  }

  Contracts {
    BOOL Time(ONTVAL time);
    DATE getTime();
  }

  Ontology { MORNING, AFTERNOON, EVENING; }
}
```

(b) The Time Context

Figure 5.1: Example MediateSpace Language Structures

We now discuss the process involved in translating conditions of arbitrary complexity to multi-dimensional spatial indexes. This process can be broken down into two main steps which we discuss in turn:

Value Mapping In order to translate our language into a spatial index we need to be able to map each part of a condition to a numeric value.

Structural Mapping We need to be able to map the structure of a condition (blocks, nested blocks, logical connectives etc) to a set of spatial indexes. This is achieved through the construction and manipulation of a *block forest*.

5.2 Value Mapping

There are three aspects of our language which require mapping to numeric values. These are as follows:

1. Map contracts to integer values to ensure that each contract maps to a specific dimension of the spatial index.
2. Map each of the data types supported by our language to a floating-point representation which can be used as min-max values along a dimension of the spatial index.
3. Map each instance of a contract to min-max values along a dimension of the spatial index.

5.2.1 Mapping Contracts to Dimensions

Each contract is mapped to an integer value, with contracts being ordered lexicographically. We assume that all nodes share the same contexts and that contexts are not added or removed during the execution of the application. Provided these assumptions are not violated a lexicographical ordering will ensure that contracts are mapped to the same dimensions on all systems.

5.2.2 Mapping Data Types to Floating-Point Values

We now discuss the mapping between each of the data types supported by the MediateSpace language and appropriate floating-point values for storage within the spatial index. This transformation is performed as follows:

Integers and Floats This transformation is straightforward, with float values mapped directly and integers mapped to their floating point counterparts.

Boolean Mapped to one or zero (one for true, zero for false).

Date Mapped according to their distance from the Unix epoch (the number of seconds elapsed since midnight on January 1st 1970 UTC/GMT).

Time Mapped according to the number of seconds that have passed since 12 midnight (00:00:00). This data type has a range from 0 to 86400, which corresponds to the number of seconds in a day.

Ontology Values Ontology values are mapped to a dimension according to their order within the context that defines them. For instance, in the Time context defined in Figure 5.1b, MORNING would be mapped to 0.0, AFTERNOON to 1.0 and EVENING to 2.0.

5.2.3 Mapping Contracts to Min-Max Values

The min-max values of the spatial index are derived by examining each contract within a condition. Contracts are either used as parameters of the Std.Compare(Contract, ComparisonOp, Value) method or they may be specified independently if the contract represents an ontology (i.e. the contract shares the name of its enclosing Context and has a single OntVal parameter). In the former case, the contract is transformed to a min or max value which depends on the ComparisonOp and Value declared within the contract. In the latter case, this transformation is performed on the value of the OntVal parameter. Note that with the exception of ontology contracts, contract parameters are not represented within the index.

In the case of ontology contracts both the min and max points are assigned to the numeric value calculated for the OntVal parameter. To specify min-max ranges for the other types we use the parameters of the Std.Compare method. This method has three parameters: the contract we are evaluating (Contract), the relational operator we are using for comparison (ComparisonOp) and a value to compare against (Value). The derived min-max values are assigned to the dimension representing the Contract. These min-max values are calculated according to the rules in Table 5.1a.

When contracts are specified using negation the *ComparisonOp* is replaced with the operator (or set of operators) which satisfy the negation. This transformation depends on the data type considered. We list each of these transformations in Table 5.1b. Note that whenever an ontology type is specified we refer to the Time ontology in Figure 5.1b (MORNING (0.0), AFTERNOON (1.0), EVENING (2.0)):

Now that we have a method for translating contracts onto our spatial index we shall demonstrate how this mapping can be achieved for conditions of arbitrary complexity.

5.3 Structure Mapping

We now discuss the process of mapping the structure of a condition (blocks, nested blocks, logical connectives etc) to a set of spatial indexes. We discuss this in terms of a system which supports only two contracts (and hence uses a two dimensional spatial index) because it simplifies the discussion. However, this method can be applied to N-contract systems without modification.

As mentioned earlier, this mapping is achieved through the construction and manipulation of a *block forest*. We begin by defining a block forest and it's operations and then discuss how it can be used to achieve our goal.

Op	Value	Min Value	Max Value
Numeric Values			
$=$	NumValue	NumValue	NumValue
$\leq$	NumValue	Unbounded	NumValue
$\geq$	NumValue	NumValue	Unbounded
$<$	NumValue	Unbounded	NumValue - ulp
$>$	NumValue	NumValue - ulp	Unbounded
Boolean Values			
$=$	true	1.0	1.0
$=$	false	0.0	0.0
Ontology Values			
$=$	AFTERNOON (1.0)	AFTERNOON (1.0)	AFTERNOON (1.0)

(a) Mapping Contracts to Min-Max Values

Op	Value	New Op	Min Value	Max Value
Numeric Values				
$\neq$	NumValue	$<$	Unbounded	NumValue - ulp
and		$>$	NumValue - ulp	Unbounded
$\not<$	NumValue	$\geq$	NumValue	Unbounded
$\not\leq$	NumValue	$>$	NumValue - ulp	Unbounded
$\not>$	NumValue	$\leq$	Unbounded	NumValue
$\not\geq$	NumValue	$<$	Unbounded	NumValue - ulp
Boolean Values				
$\neq$	true	$=$	0.0	0.0
$\neq$	false	$=$	1.0	1.0
Ontology Values				
$\neq$	AFTERNOON (1.0)	$=$	MORNING (0.0)	MORNING (0.0)
or		$=$	EVENING (2.0)	EVENING (2.0)

(b) Mapping Negated Contracts to Min-Max Values

Table 5.1: Rules for Mapping contracts to Min-Max Values

5.3.1 Block Forests

Indexes are generated by parsing conditions and building what we have termed block forests. In the building of a block forest we distinguish between *outer blocks* and *nested blocks*, where nested blocks refer to blocks contained within other blocks and outer blocks refer to blocks without this constraint. A block forest is a forest of trees with the following properties:

- Each node reflects a single block (outer or nested).
- Root nodes represent outer blocks
- Child nodes represent nested blocks.
- Each node stores an arbitrary number of indexes.

The block forest has two operations which may be performed on it: *merging* and *copying*. Both of these operations are applied to a pair of blocks and return a single block.

Merging involves obtaining the cartesian product of some subset of the indexes in both blocks and combining them together, ensuring that all indexes are as small as possible. That is, we should combine each pair of indexes, reducing the volume of the index rectangle whenever possible. Dimensions of an index are only merged if they have been modified (i.e. are not null). If the sets of modified dimensions of the two indexes being merged are disjoint a merge can be achieved simply by copying the modified dimensions from one block to the other. If, however, the sets of modified dimensions are not disjoint we must combine the shared dimensions with one another, reducing the range of each dimension whenever possible.

Copying simply involves moving a set of indexes from one block to another without modification.

When the processing of a condition is complete we will be left with a single block which contains all of the indexes pertinent to the condition.

5.3.2 Manipulating Block Trees

As discussed, contracts are mapped to dimensions of an index and we can generate min-max values to apply to a dimension whenever an instance of a given contract is seen within a condition. These values depend on the operator used and the data type.

Contracts may be connected using conjunctions or disjunctions and may be grouped using blocks and nested blocks to arbitrary levels. The number of indexes created per condition varies depending on the structure of the condition. The algorithm assumes that conjunctions have a higher precedence than disjunctions.

We now discuss each of these structural elements with examples, ranging from the very simple to those consisting of many Contracts and several nested blocks. Each example includes a condition, a graph representing the indexes created and a diagram illustrating the building of the block tree during execu-

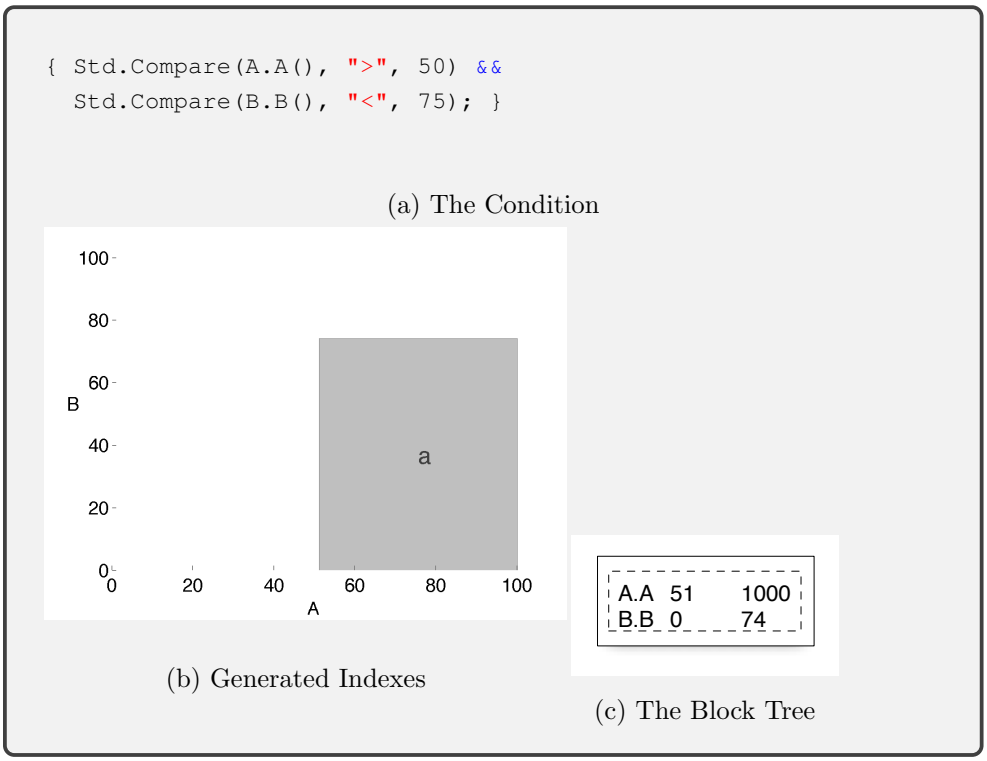


Figure 5.2: A Simple Conjunction

tion. The contracts A.A() and B.B() are used throughout the examples and for simplicity we assume that in every case both contracts are parameterless and return an integer value. We also assume that the minimum and maximum values for a dimension are 0 and 1000 respectively, with dimensions defaulting to these values if no other value is specified.

5.3.2.1 Conjunctions

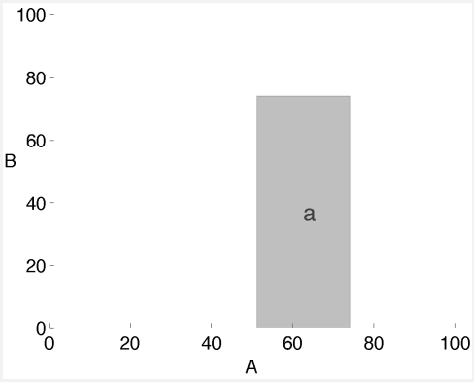
We begin with the simple case of two contracts connected using a conjunction. This can be represented using a single rectangle as illustrated in Figure 5.2. Figure 5.3 illustrates the more complicated case where both upper and lower bounds are expressed for contract A.A which tightens the rectangle but still only requires a single index.

5.3.2.2 Disjunctions

When disjunctions are used additional indexes must be created to represent these alternative branches of the condition. This is demonstrated in Figure 5.4, where a second index has been created to represent the constraints of the

```
{ Std.Compare(A.A(), ">", 50) &&
  Std.Compare(A.A(), "<", 75) &&
  Std.Compare(B.B(), "<", 75); }
```

(a) The Condition



(b) Generated Indexes

A.A	51	74
B.B	0	74

(c) The Block Tree

Figure 5.3: A Simple Conjunction: Fully Constraining A.A()

B.B contract. Intersecting with either of these indexes is sufficient to include the associated message on the candidate list.

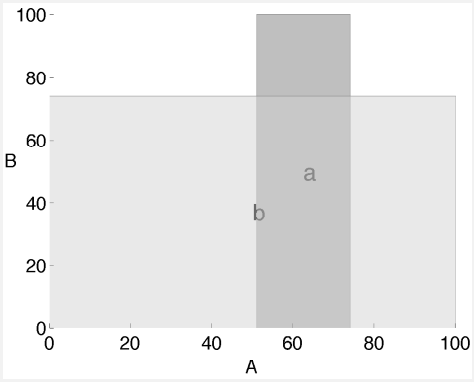
5.3.2.3 Outer Blocks

Blocks (both outer and nested) can also be connected using conjunctions and disjunctions. Outer blocks connected with a conjunction should be merged using the full cartesian product of the indexes in both blocks. When the merge is complete, the right-hand-side (RHS) block should be removed and the left-hand-side (LHS) should be replaced with the newly created merged block. Figure 5.5 illustrates this, showing the process and result of merging the pairs (Node1_Index1, Node2_Index1) and (Node1_Index1, Node2_Index2).

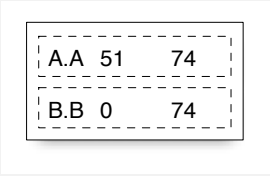
Outer blocks connected via a disjunction should be handled by copying all of the indexes from the RHS block into the LHS block. This is sufficient because the two blocks are independent. Figure 5.6 illustrates this simple process by copying the RHS indexes into the LHS.

```
{ Std.Compare(A.A(), ">", 50) &&
  Std.Compare(A.A(), "<", 75) ||
  Std.Compare(B.B(), "<", 75) }
```

(a) The Condition



(b) Generated Indexes



(c) The Block Tree

Figure 5.4: A Simple Disjunction

5.3.2.4 Nested Blocks

Figures 5.7, 5.8 and 5.9 all illustrate how the indexing algorithm handles the nesting of blocks. For each nested block, a child node is added to the block tree underneath the tree node representing its containing block. After a nested block has been processed its block node is combined with its parent. Again, this involves either a merge or copy depending on whether the nodes were connected via a conjunction or disjunction respectively.

However, instead of using the full cartesian product during a merge, only the final index of the parent block is included in product generation.

In Figure 5.7 the combination is a straightforward merge because the nested block contains only a single index. Thus, the final combined block can be created simply by merging the last index in the parent block with the index in the child.

The merge of the nested block in Figure 5.8 is more involved as the child contains two indexes. The final combined block is created by generating the cartesian product of the child indexes and the last index of the parent, generating the product: (Node1_Index2, Node2_Index1) and (Node1_Index2, Node2_Index2).

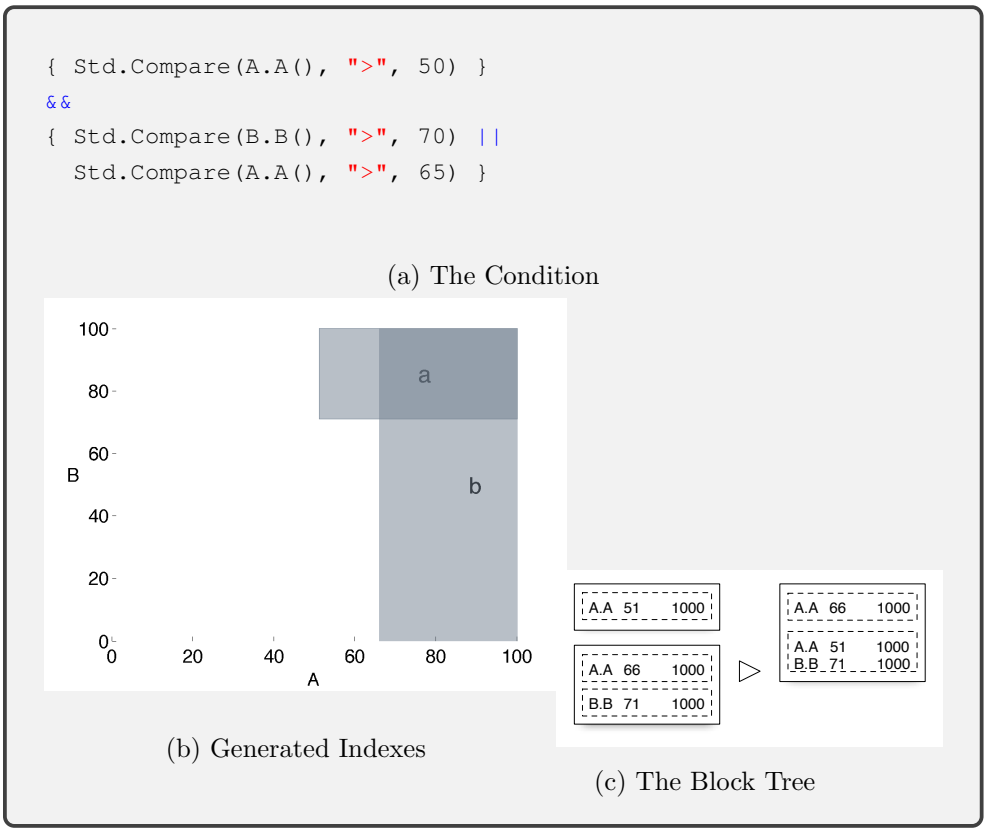


Figure 5.5: Merging Two Blocks (Conjunction)

Figure 5.9 features two nested blocks and operates similarly to the previous example, but with the difference that the second nested block is connected to it's parent via a disjunction. Thus, when this nested block has been processed, it will be copied into it's parent block, which in turn is merged with the top-level block to form the final block.

5.3.2.5 Handling Repeated Contracts and Parameters

If the same contract is specified multiple times within one condition and both are stored within the same index (either because a conjunction connects the two or they are both members of either side of a merge) it may be possible to reduce the size of one or more indexes. We call this "spatial reduction" for brevity. For instance, consider the condition in Figure 5.10a where the first two contracts constrain the index dimension for A.A to between the values of 51.0 and 74.0 (assuming that A.A returns an integer value). The third condition gives us additional information about the value of A.A; namely that A.A should also have a value which is greater than 60.0. Thus, we can further

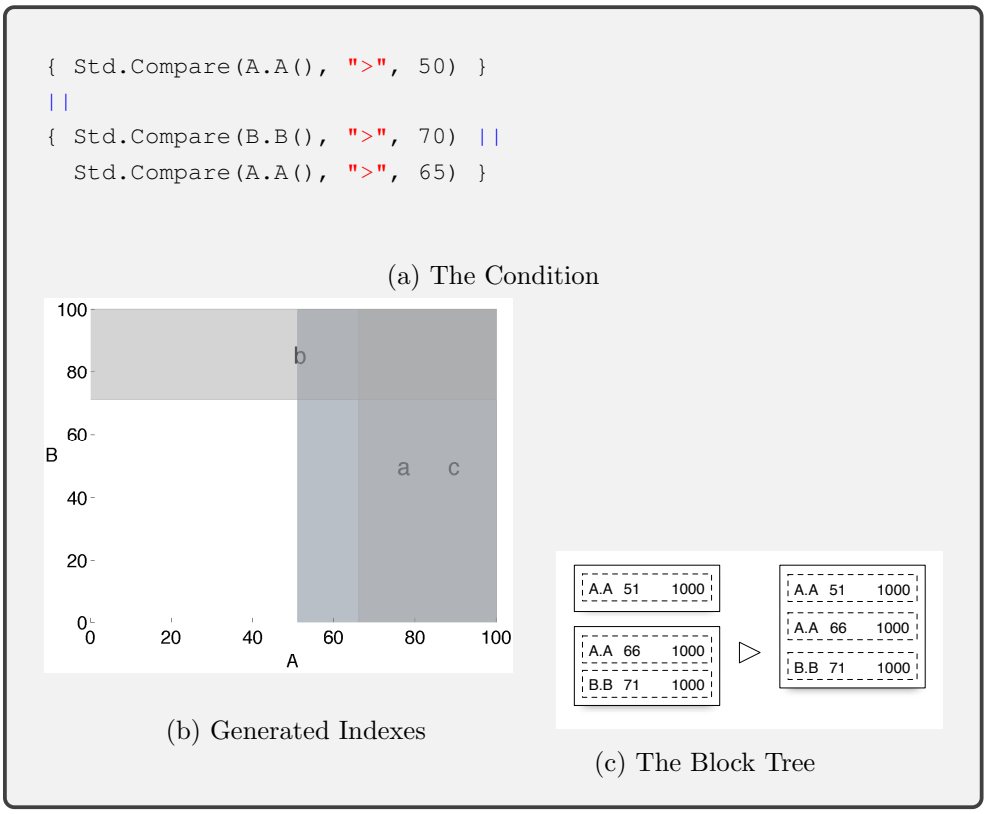


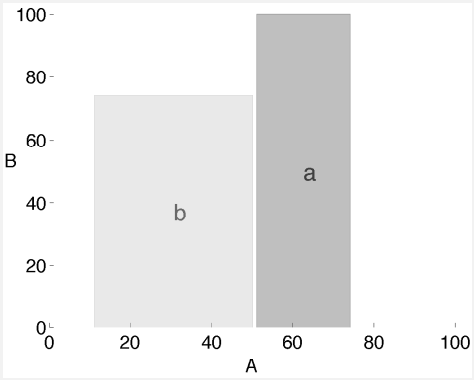
Figure 5.6: Copying Two Blocks (Disjunction)

constrain the index dimension to values between 61.0 and 74.0. The second condition in Figure 5.10b will result in the creation of the same index but will be achieved through a merge operation between the nested block and its parent. Although the condition can simply be rewritten to achieve the same meaning spatial reduction becomes useful when we introduce parameterised contracts.

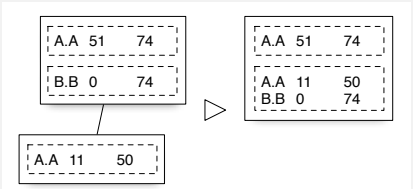
The spatial reduction method remains valid when contracts may be parameterised. This is because each index can only contain contracts which must all be true (i.e. they are all connected, directly or indirectly with a conjunction). Thus, even if we have a condition which contains the same contract twice but with different parameter values it is safe to reduce the relevant index dimension because both contracts must be fulfilled. For example, if we have a message with the condition in Figure 5.10c (which reduces the Temperature dimension of the index to be greater than 80.0) then applying the query in Figure 5.10d will safely disregard this message because its condition cannot be fulfilled regardless of the parameter values used by the contracts within the query. Note that as parameters are not considered during index genera-

```
{ Std.Compare(A.A(), ">", 50) &&
  Std.Compare(A.A(), "<", 75) ||
  Std.Compare(B.B(), "<", 75) &&
  {
    Std.Compare(A.A(), ">", 10)
    &&
    Std.Compare(A.A(), "<=", 50)
  }
}
```

(a) The Condition



(b) Generated Indexes



(c) The Block Tree

Figure 5.7: Simple Nested Block Merging

tion they can be safely excluded from MessageRequests without affecting the operation of the algorithm.

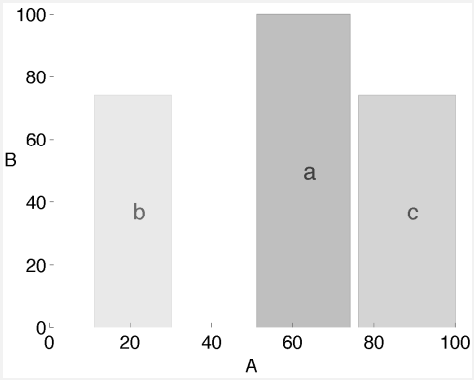
5.3.2.6 Handling All (∀) and Exists (∃)

MediateSpace supports a modified form of the ∀ and ∃ quantification operators (discussed in Section 4.2.1). These allow us to express that all conditions must be true (∀) or that at least n and at most m of the conditions are true (∃ n .. m).

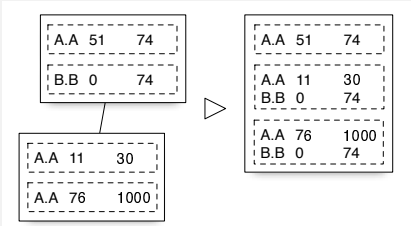
∀ ("forall") conditions are trivial to represent as the only stipulation made is that all the conditions must be true. Hence, it is sufficient to simply construct a compound expression where each condition is joined with a logical AND.

```
{ Std.Compare(A.A(), ">", 50) &&
  Std.Compare(A.A(), "<", 75) ||
  Std.Compare(B.B(), "<", 75) &&
  {
    Std.Compare(A.A(), ">", 10)
    &&
    Std.Compare(A.A(), "<=" 30)
    ||
    Std.Compare(A.A(), ">", 75)
  }
}
```

(a) The Condition



(b) Generated Indexes

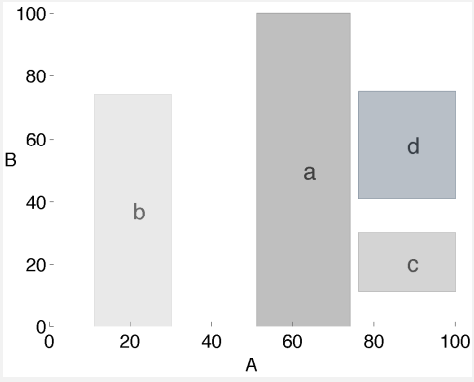


(c) The Block Tree

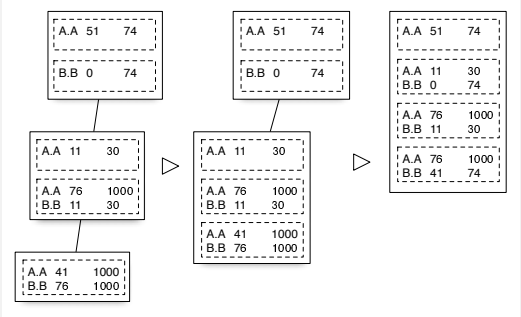
Figure 5.8: Cartesian Merging of Nested Blocks

```
{ Std.Compare(A.A(), ">", 50) &&
  Std.Compare(A.A(), "<", 75) ||
  Std.Compare(B.B(), "<", 75) &&
  {
    Std.Compare(A.A(), ">", 10)
    &&
    Std.Compare(A.A(), "<=" 30)
    ||
    Std.Compare(A.A(), ">", 75)
    &&
    Std.Compare(B.B(), ">", 10)
    &&
    Std.Compare(B.B(), "<=", 30)
    ||
    {
      Std.Compare(B.B(), ">", 40)
      &&
      Std.Compare(A.A(), ">", 75)
    }
  }
}
```

(a) The Condition



(b) Generated Indexes



(c) The Block Tree

Figure 5.9: Cartesian Merging and Block Copying

```
{ Std.Compare(A.A(), ">", 50) &&
  Std.Compare(A.A(), "<", 75) &&
  Std.Compare(A.A(), ">", 60) }
```

(a) A Simple Example of Repeated Contracts

```
{ Std.Compare(A.A(), ">", 50) &&
  Std.Compare(A.A(), "<", 75) &&
  {
   Std.Compare(A.A(), ">", 60)
  }
}
```

(b) An Example of Repeated Contracts with Nesting

```
{ Std.Compare(Temperature.getTemp("london"), ">", 50) &&
  Std.Compare(Temperature.getTemp("brighton"), ">", 80) }
```

(c) An Example of Parameterised Repeated Contracts

```
{ Std.Compare(Temperature.getTemp(...), ">", 45) &&
  Std.Compare(Temperature.getTemp(...), "<", 75) }
```

(d) An Example Query to be applied to the above condition

Figure 5.10: Parameterised and Non-parameterised Repeated Contracts

For example, ∀ A, B, C can be represented as A && B && C. This can be easily transformed to a single index using the procedures discussed above.

Although in the simplest case (when ∃ (1, numConds)) ∃ conditions are equally as trivial to represent as ∀ conditions (simply joining each condition with a logical OR), in general the task is more complicated because of the inclusion of minimum and maximum parameters. The problem can be broken down into two parts:

1. constructing conditions which satisfy the minimum parameter, and
2. constructing conditions which satisfy the maximum parameter

Satisfying the minimum parameter (n) requires the generation of all combinations of conditions of size n. The elements of each combination should be connected with a conjunction, and each combination should be connected with a disjunction.

Satisfying the maximum parameter (m) requires that two extensions are made to this algorithm. Firstly, combinations must be generated of all condi-

```
Exists(2,3) A, B, C, D
```

(a) An example ∃ condition

```
Given the conditions {A, B, C, D},

CALCULATING N

If n = 2, we would generate the following combinations:

  {(A,B), (A,C), (A,D) (B,C), (B,D), (C,D)}

The resulting condition would be:

  (A && B) || (A && C) || (A && D) || (B && C) ||
  (B && D) || (C && D)

CALCULATING M

If m = 3, the resulting compound condition would become:

  (A && B && !C && !D) || (A && C && !B && !D) ||
  (A && D && !B && !C) || (B && C && !A && !D) ||
  (B && D && !A && !C) || (C && D && !A && !B) ||
  (A && B && C && !D) ||  (A && B && D && !C) ||
  (A && C && D && !B) ||  (B && C && D && !A)
```

(b) An Application of the ∃ algorithm to the given example

Figure 5.11: Transforming ∃ blocks to a form using logical connectives

tions of sizes n to m, with each of these combinations being logically connected as discussed in the previous paragraph. Secondly, we must ensure that no more than m conditions can be resolved as true and still satisfy the generated compound condition. To this end, we must extend each group of conjunctions with additional conjunctions stipulating that all other conditions must NOT be true. We use this algorithm to translate ∃ blocks into a form readily useable by our indexing algorithms discussed above. See Figure 5.11 for a example application of this algorithm.

However, in cases where a large number of conditions are used this method can result in the production of an extremely large number of indexes. To counteract this issue we provide an algorithm for continually simplifying the

∃ block until the number of indexes produced falls below a defined number.

Our simplification algorithm continually decrements the minimum value specified for the ∃ block (n) until the number of indexes to generate falls below the defined cap. Both the minimum and maximum values of the ∃ block are then set to this value and stage two of the translation algorithm defined above is not applied. This provides a reasonable approximation while keeping the number of indexes as low as possible. Importantly, although this may result in some false positives it will never cause false negatives to occur.

An optimisation can be applied when the maximum value (m) equals the number of conditions in the ∃ block. In this case, it is not possible for more than m conditions to be resolved as true. Therefore, we need only require that the minimum number of conditions be satisfied and can disable stage two of the translation algorithm.

5.3.3 Detecting Ill-Formed Conditions

It is possible to write conditions that can never succeed. The following condition can never succeed because a context value cannot be greater than 80.0 and less than 50.0 at the same time:

```
{ Std.Compare(A.A(), ">", 80) && Std.Compare(A.A(), "<", 50); }
```

Our index generating algorithm can be used to detect such ill-formed conditions by examining each generated index for any dimension where min > max. If this property holds for any dimension then we can conclude that the condition is ill-formed.

5.4 Summary

In the preceding chapter we discussed our methodology for mapping contextual conditions to spatial indexes. This allows us to index our conditions, supporting the efficient lookup of the information associated with them.

The lookup phase allows us to restrict the set of conditions we need to consider, but it is still necessary to evaluate the conditions to ensure that the requesting party holds the necessary context to receive the associated content. This is the focus of the next chapter where we discuss how our contextual conditions can be translated into an OWL representation. The conditions can then be evaluated using an OWL reasoner.

6 Context Reasoning Using OWL

6.1 Introduction

The MediateSpace language allows the construction of contextual conditions consisting of an arbitrary number of contracts. These contracts may be connected using logical operators (&&, ||) and quantification operators ($\forall$, $\exists$), and conditions can contain an arbitrary number of blocks nested to any level. In order to evaluate these conditions we have chosen to use the OWL language[1] which we briefly discuss in the next section. We then go on to describe the MediateSpace OWL ontology, which provides the vocabulary used when translating the MediateSpace language to its OWL representation. We then discuss the MediateSpace Evaluation ontology which is generated to represent concrete instances of our language and demonstrate how these structures can be used for evaluation.

6.2 The OWL Language

This section briefly discusses the basics of working with OWL, the various syntaxes which are available for representing it and the main assumptions made during the reasoning process. For additional discussion, see Section 3.6.7.

6.2.1 Classes, Individuals and Properties

The Web Ontology Language (OWL) supports the modelling of information domains through the specification of classes and individuals. Classes provide a model for a type of object, which may be tangible or just conceptual in nature. For example, in a botanical ontology there may be a class of type Rose. Individuals refer to concrete instances of a class, so in our botanical

[1] http://www.w3.org/TR/owl-features/

ontology multiple individuals may be created which each represent a single rose.

Classes are defined to accept a desired subset of individuals within a domain (such as the Rose class above which accepts all individuals which are roses). This subset of individuals is defined in part through the use of object and data properties, which allow us to establish relationships with individuals of a given class (or set of classes) or with a given literal value (or range of values) respectively. The relationships established for a class can be made arbitrarily complex by combining properties using intersections and unions. For example, a Rose class could stipulate that all roses have between n and m petals and also have a stalk. In this example, we have restricted membership of this class to only those individuals with between n and m petals and one stalk.

6.2.2 Syntaxes

A variety of OWL syntaxes are available, including Turtle[2], OWLXML and the Manchester syntax [52]. We focus on the Manchester syntax because of its focus on readability, terseness and support for users without a background in description logics. As discussed above, OWL classes of arbitrary complexity can be defined using intersections, unions and properties, and these can all be represented in the Manchester syntax. We provide a summary of the pertinent parts of this syntax in Tables 6.1 and 6.2 with a mapping to its OWLXML representation. Note that the "and" and "or" operators can be mapped directly to their counterparts in the MediateSpace language.

6.2.3 Reasoning Assumptions

OWL has a number of unusual properties which affects how it may be used. For instance, most systems in use today use the closed world assumption which posits that if a piece of information is not found within the system then it can be concluded that it does not exist. For example, an accounting system may conclude that if a given customer number is not found then the company does not have a customer with that number. OWL uses the open world assumption which would not make this conclusion. Instead, the existence of this customer number would remain undecided unless it was explicitly stated that the given customer number does not exist. By remaining open OWL ensures that the conclusions it makes are consistent at all times. In our accounting example, if the desired customer number was later added to the ontology we could now conclude that the customer is present without contradicting any earlier conclusions. This open world viewpoint can cause issues when attempting to specify negative conditions. For example, if we were to express the class of Male as "not Female" we might expect that any individual not classified as

[2]http://www.w3.org/TeamSubmission/turtle/

Manchester Syntax	OWLXML Syntax

`AClass1 and AClass2`	`<ObjectIntersectionOf>`
	`  <Class IRI="#AClass1"/>`
	`  <Class IRI="#AClass2"/>`
	`</ObjectIntersectionOf>`

`AClass1 or AClass2`	`<ObjectUnionOf>`
	`  <Class IRI="#AClass1"/>`
	`  <Class IRI="#AClass2"/>`
	`</ObjectUnionOf>`

`Individual: #indAClass1`
`  Types: #AClass1`
`Individual: #indAClass2`
`  Types: #AClass2`
	`<ClassAssertion>`
	`  <Class IRI="#AClass1"/>`
	`  <NamedIndividual IRI="#indAClass1"/>`
	`</ClassAssertion>`
	`<ClassAssertion>`
	`  <Class IRI="#AClass2"/>`
	`  <NamedIndividual IRI="#indAClass2"/>`
	`</ClassAssertion>`

Table 6.1: Classes and Individuals : Manchester Syntax to OWL XML

Female to be implicitly classified as Male. This conclusion will not be made however as the individual may be classified as Female at some point in the future, so to declare them Male now may lead to a contradiction later on. This proved a challenge when attempting to provide support for negated conditions in the MediateSpace language. We discuss this further and provide a solution in Section 6.5.2.5.

In addition, OWL does not provide support for the unique name assumption, meaning that it is permissible for more than one name to refer to the same entity. For example, the King's Cross train station may be referred to using the names kingsCross or train_station_kings_cross.

For our system we chose to use classes to model the contextual conditions and individuals to represent the context values present within a node's tuple space. Through the application of an OWL reasoner we can then evaluate conditions. We shall now briefly motivate our use of the OWL language. This

Manchester Syntax	OWLXML Syntax

`hasObjValue some AClass1`	`<ObjectSomeValuesFrom>`
	`  <ObjectProperty IRI="#hasObjValue"/>`
	`  <Class IRI="#AClass1"/>`
	`</ObjectSomeValuesFrom>`

`hasDataValue "5"^^xsd:integer`	`<DataHasValue>`
	`  <DataProperty IRI="#hasDataValue"/>`
	`  <Literal datatypeIRI="&xsd;integer">`
	`  5`
	`  </Literal>`
	`</DataHasValue>`

`hasDataValue some`
`  integer[<= "25"^^xsd:integer]`
	`<DataSomeValuesFrom>`
	`  <DataProperty IRI="#hasDataValue"/>`
	`  <DatatypeRestriction>`
	`  <Datatype`
	`  abbreviatedIRI="xsd:integer"/>`
	`  <FacetRestriction`
	`  facet="&xsd;maxInclusive">`
	`  <Literal`
	`  datatypeIRI="&xsd;integer">25`
	`  </Literal>`
	`  </FacetRestriction>`
	`  </DatatypeRestriction>`
	`</DataSomeValuesFrom>`

`Individual: #indAClass1`
 	
`  Types: #AClass1`
 	
`  Facts:`
`  hasObjValue #indAClass2,`
`  hasDataValue 5`
	`<ObjectPropertyAssertion>`
	`  <ObjectProperty IRI="#hasObjValue"/>`
	`  <NamedIndividual IRI="#indAClass1"/>`
	`  <NamedIndividual IRI="#indAClass2"/>`
	`</ObjectPropertyAssertion>`
	`<DataPropertyAssertion>`
	`  <DataProperty IRI="#hasDataValue"/>`
	`  <NamedIndividual IRI="#indAClass1"/>`
	`  <Literal datatypeIRI="&xsd;integer">`
	`  5`
	`  </Literal>`
	`</DataPropertyAssertion>`

Table 6.2: Properties : Manchester Syntax to OWL XML

is followed by a description of the representation and methodology used to map our MediateSpace language to OWL.

6.3 Motivation

Although we could have chosen to implement a specialised reasoner to evaluate our MediateSpace language we decided upon using the OWL language. This was for a number of reasons:

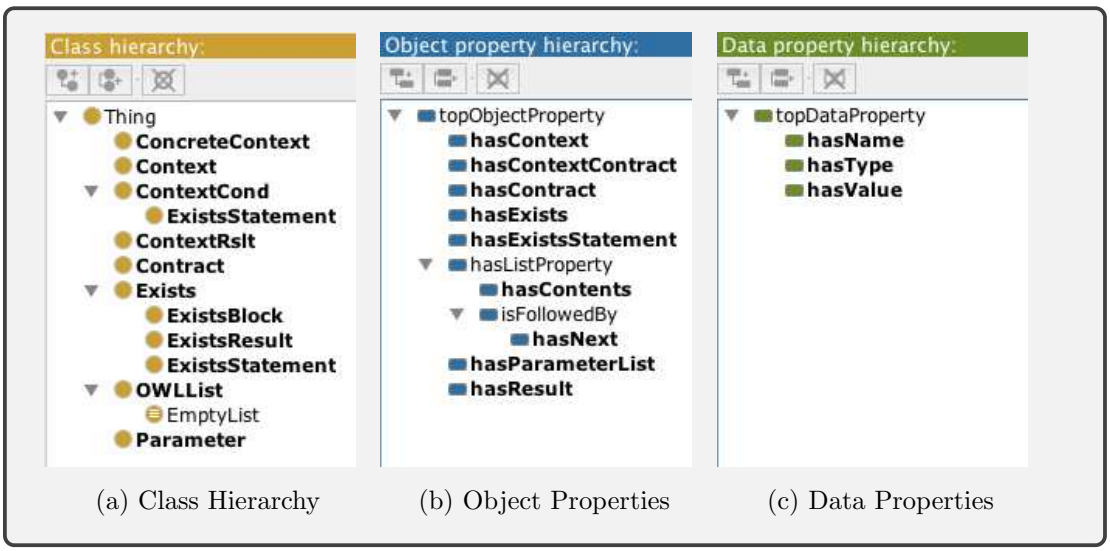
- The OWL language and description logics in general are still actively researched and applied to a number of use cases. Thus, we will be able to reap the benefits of future research which will potentially improve the tractability of reasoning.
- OWL is a mature specification with a significant number of reasoner implementations available for a wide variety of hardware and operating systems. Our implementation will benefit from future improvements to these implementations. For example, future releases could provide optimisations for runtime efficiency or memory consumption. Reasoners have also been written for use within larger applications such as the Oracle database.
- A number of tools are available that ease the development and debugging of OWL implementations. For example, Protege[3] and SWOOP[4]. There are also libraries for manipulating OWL within other languages. For instance, Thea[5] allows Prolog programs to process OWL ontologies and the OWL API[6] provides a Java interface to build, manipulate and evaluate OWL ontologies on a variety of supported reasoners.
- The OWL specification[7] supports ontology versioning and extension, meaning that our language can be safely amended and extended without causing issues for consumers of our ontology. The specification also provides support for interoperability between ontologies, meaning that other languages could be translated into the MediateSpace language for evaluation or vice versa.

6.4 MediateSpace OWL Ontology

The MediateSpace OWL Ontology provides the vocabulary used for representing the MediateSpace language in OWL. The ontology is available at http://www.dmatthews.co.uk/ontologies/mediate-space/1.0/mediate-space.owl and its contents are summarised in Figure 6.1.

[3] http://protege.stanford.edu
[4] https://code.google.com/p/swoop/
[5] http://www.semanticweb.gr/thea/
[6] http://owlapi.sourceforge.net/
[7] http://www.w3.org/TR/webont-req/

A brief description of the classes provided and how they are used to translate our language into OWL is provided in Table 6.3.



(a) Class Hierarchy (b) Object Properties (c) Data Properties

Figure 6.1: OWL Class and Property Hierarchies (in Protege[8])

6.5 MediateSpace Evaluation Ontology

The MediateSpace language is transformed into the OWL Manchester syntax [52], which is then loaded into the OWL API [51] within our Java program for evaluation.

We use the FaCT++ reasoner [85], which is written in C++ and allows the speedy evaluation of OWL ontologies, connecting the reasoner to our java program via a JNI binding. We chose the FaCT++ reasoner because it is actively maintained, runs natively on the machine and is straightforward to use as it can be manipulated directly using the OWL API.

We now discuss the representation of each MediateSpace language element within the OWL Manchester notation.

6.5.1 Representing Contexts and Concrete Contexts

A single subclass of Context is created for every Context within the local tuple space and a subclass of Contract is created for every contract within each context.

When participant nodes bind to a regional node they send a message containing all of the ConcreteContext tuples they support. Each regional node

[8] http://protege.stanford.edu

Class	Description
Context	Representing Context structures
Contract	Representing a contract within a Context structure
ConcreteContext	Representing ConcreteContext Structures
ContextCond	Representing context conditions
ContextRslt	Representing the individual contracts within a condition and nested blocks
Parameter	Representing the values within contract parameters and context value results. Structured as nested lists using the OWL-List class defined by [30]
Exists Classes	For handling the representation and evaluation of ∃ blocks. For efficiency, the conditions within ∃ blocks are evaluated once prior to the evaluation of the other conditions.

Table 6.3: A Summary of the MediateSpace OWL Ontology Classes

generates a ConcreteContext subclass for each of these received tuples, with each generated class subclassing the parent Context and ConcreteContext classes discussed in the previous paragraph. A single individual is also created for each of these subclasses to allow individuals to refer to these classes within their definitions. This structure is summarised in Figure 6.2.

As discussed in previous chapters, Participant nodes can request context values from the Regional node they are bound to. The Regional node is then responsible for establishing which of their bound Participant nodes (if any) can fulfil the request (i.e. which Participant registered a ConcreteContext tuple for the appropriate Context with the Regional node). This task can be achieved by querying the generated OWL structure to obtain a list of viable participants. This is achieved in three steps:

1. Build a query class of the form: `contextName and ConcreteContext`,
2. Insert the query class into the ontology and obtain a list of all subclasses of this query,
3. Choose the most appropriate subclass according to some criteria,
4. Map the subclass against the appropriate tuple in the Regional tuple space using the subclass's IRI.

```
context_A
  subclass:
    Context
```

(a) Structure of a Context class

```
concreteContext_A_0
  subclass:
    context_A and
    ConcreteContext
```

(b) Structure of a ConcreteContext class

```
indi_context_A
  Type:
    context_A
```

(c) Structure of a Context individual

```
indi_contract_A1
  Type:
    Contract
```

(d) Structure of a Contract individual

Figure 6.2: Context and ConcreteContext Forms

At present, step 3 is handled simply by choosing a subclass at random, but the OWL representation could be extended to support more intelligent behaviour. For example, if a context value is only deemed valid within a certain distance of the requesting participant (location information loses it's utility as distance grows for instance), a data property could be used to specify the distance of the bound Participant from the Regional node. The query could then be amended to include this constraint and only retrieve subclasses within the valid range.

6.5.2 Representing and Evaluating Conditions

Contextual conditions are represented as a combination of classes, properties and literals. Figure 6.3 illustrates it's basic structure. Each condition has a root *ContextCond* class with an arbitrary number of *hasContextContract* object properties. Each *hasContextContract* property has a *ContextRslt* class as its range, with each *ContextRslt* representing a single contract within the condition. Each *ContextRslt* specifies the Context name, Contract name, parameter list and expected result values/ranges.

All supported properties are summarised in Table 6.4. Note that both the parameter and result lists are structured as nested lists so that a list can itself be specified as a parameter if desired.

Our transformation algorithm allows all types of conditions to be represented in OWL. To achieve this we have defined six representations to be used as appropriate when transforming a condition. These are *Exact Match*, *Range*, *Ontology*, *Nested Blocks*, *Negated Contracts* and *Exists (∃)*. These will now each be discussed in turn.

Property	Description	Domain	Range
hasContextContract	One created for each contract within the condition. They are also created between all ContextRslt individuals to support nested blocks.	ContextCond ContextRslt	ContextRslt
hasContext	Used to indicate which Context structure this ContextRslt refers to. A single class and individual exist for each Context.	ContextRslt	Context
hasContract	Used to indicate which Contract structure this ContextRslt refers to. A single class and individual exist for each Contract.	ContextRslt	Contract
hasParameterList	Refers to the list of parameters given for this contract. If the contract does not have parameters this property is omitted.	ContextRslt	OWLList
hasExists	**Class Expression:** Refers to the ExistsResult class used to determine if an ∃ block has passed evaluation. **Individual:** Indicates that an ∃ block has passed evaluation by linking to an ExistsResult individual.	ContextCond	ExistsResult
hasExistsStatement	Refers to a contract forming part of an ∃ block.	ExistsBlock	ExistsStatement
hasResult	**Class Expression:** Refers to the list of values that must be matched for this contract to succeed. **Individual:** Refers to the list of values obtained by executing this contract.	ContextRslt	OWLList
hasValue	Refers to the value given for a particular parameter or result value. If used to represent a result value, this can also be expressed as a value range (e.g. ≥ 50, ≤ 75).	Parameter	Literal
hasType	Refers to the datatype used to represent the given parameter or result value.	Parameter	StringLiteral
hasName	**Class Expression:** Refers to the string literal to be matched by an ExistsResult individual for the ∃ block to pass evaluation. **Individual:** Refers to the string literal used to identify a successfully evaluated ∃ block.	ExistsResult	StringLiteral

Table 6.4: A Summary of the MediateSpace OWL Ontology Properties

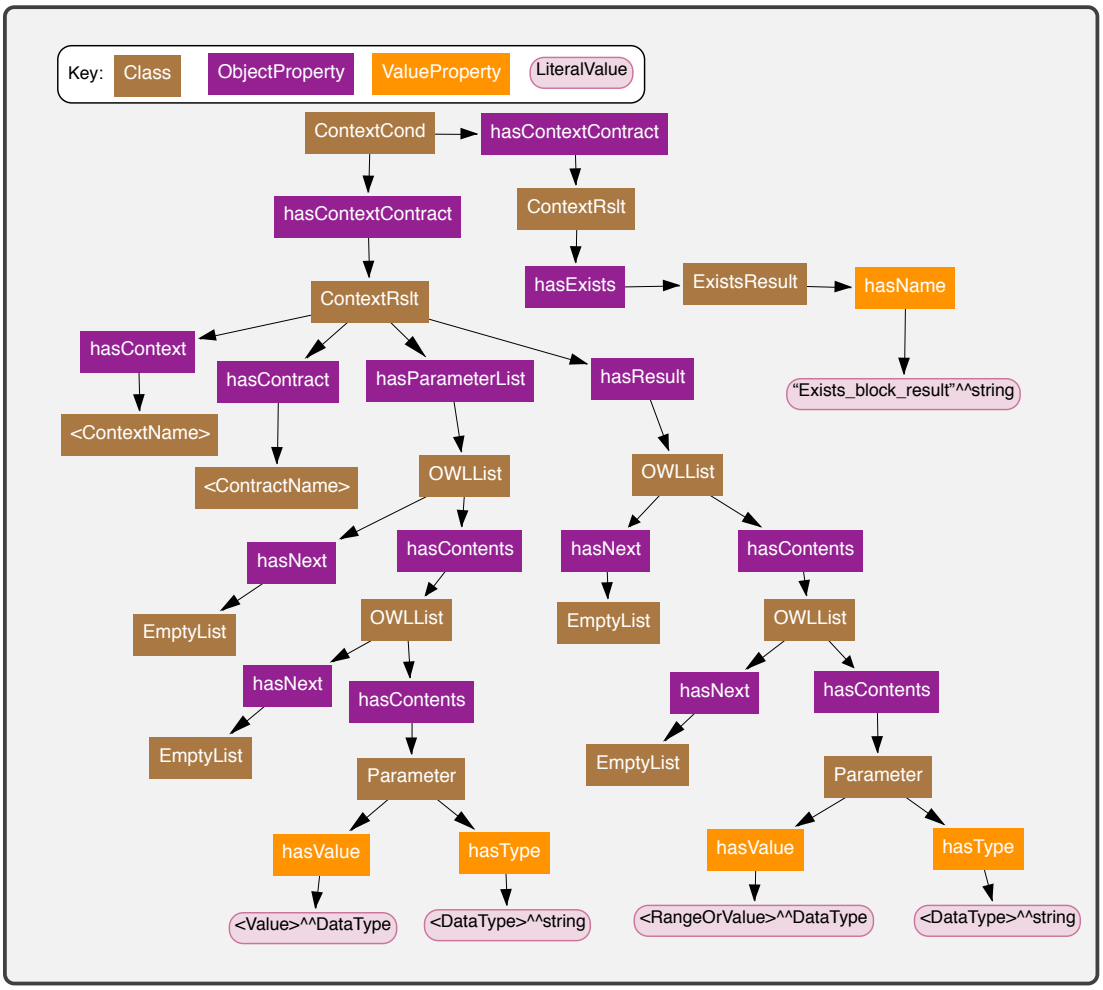


Figure 6.3: Representing a Contextual Condition

6.5.2.1 Representing Exact Matches

Figure 6.4 illustrates how to represent exact matching, with the condition succeeding if the result of the contract A.A1() is the date equal to exactly 7 PM on 27/06/2014. The contract has a Date return type and this is represented explicitly via the hasType property, and is specified as the value type to ensure that OWL processes the value correctly.

6.5.2.2 Representing Range Matches

Figure 6.5 illustrates how range conditions can be expressed. Multiple ranges can be specified for a contract within one ContextRslt section which helps to reduce space requirements and improves readability.

```
{ Std.Compare(A.A(), ==, 27/06/2014:19:00:00) }
```

(a) The Condition

```
ContextCond
 and (hasContextContract some
    (ContextRslt
     and (hasContext some A_Context)
     and (hasContract some A1_Contract)
     and (hasResult some
       (OWLList
        and (hasContents some
           (OWLList
            and (hasContents some
              (Parameter
               and (hasType value "xsd:dateTime"^^string)
               and (hasValue value "2014-06-27T19:00:00-00:00"^^dateTime)))
            and (hasNext some EmptyList)))
         and (hasNext some EmptyList)))))
```

(b) The OWL Code

Figure 6.4: An exact match (==) condition

6.5.2.3 Representing Ontology Matches

Figure 6.6 illustrates ontology matching, where the result is expressed as a floating-point value. This value represents the position of the ontology concept to be matched within its Context structure. For example, in Figure 6.6 the condition should succeed when the ontology concept A_1 is matched. Position is zero-indexed and A_1 is the second concept defined within the A Context structure so the value 1.0 is chosen.

This example also illustrates how parameters are mapped to its OWL representation.

6.5.2.4 Representing Nested Blocks

Nested blocks are modelled simply by connecting parent and child blocks via a hasContextContract object property. This is illustrated in Figure 6.7.

6.5.2.5 Representing Negated Conditions

The MediateSpace language allows contracts to be negated. For instance, we may wish to evaluate the following negated condition:

```
{ Std.Compare(A.A2(), >, 50) && Std.Compare(A.A2(), <, 75) &&
  Std.Compare(B.B1(), <, 75); }
```

(a) The Condition

```
ContextCond
 and (hasContextContract some
    (ContextRslt
     and (hasContext some A_Context)
     and (hasContract some A2_Contract)
     and (hasResult some
      (OWLList
       and (hasContents some
          (OWLList
           and (hasContents some
            (Parameter
             and (hasValue some integer[< "75"^^integer, > "50"^^integer])
             and (hasType value "xsd:integer"^^string)))
           and (hasNext some EmptyList)))
       and (hasNext some EmptyList)))))
 and (hasContextContract some
    (ContextRslt
     and (hasContext some B_Context)
     and (hasContract some B1_Contract)
     and (hasResult some
        (OWLList
         and (hasContents some
            (OWLList
             and (hasContents some
              (Parameter
               and (hasValue some integer[< "75"^^integer])
               and (hasType value "xsd:integer"^^string)))
             and (hasNext some EmptyList)))
         and (hasNext some EmptyList)))))
```

(b) The OWL Code

Figure 6.5: Two range conditions, with dual ranges on A.A2()

```
{ A.A(A_1); }
```

(a) The Condition

```
ContextCond
 and (hasContextContract some (ContextRslt
     and (hasContext some A_Context) and (hasContract some A_Contract)
     and (hasParameterList some (OWLList
         and (hasContents some
            (OWLList
             and (hasContents some
                (Parameter
                 and (hasType value "xsd:string"^^string)
                 and (hasValue value "A_1"^^string)))
             and (hasNext some EmptyList)))
         and (hasNext some EmptyList)))
     and (hasResult some (OWLList
         and (hasContents some
            (OWLList
             and (hasContents some
                (Parameter
                 and (hasType value "xsd:double"^^string)
                 and (hasValue value "1.0"^^double)))
             and (hasNext some EmptyList)))
         and (hasNext some EmptyList)))))
```

(b) The OWL Code

Figure 6.6: An Ontology Condition

```
{ !B.B(B_1) && !Std.Compare(B.B1(), "<=", 45); }
```

Although OWL allows the use of negation the open world assumption and lack of unique name assumption makes it difficult to infer the semantics we want without requiring the addition of many additional axioms. We chose to handle this by transforming the negated contexts into positive forms (using a similar method to that used for negated conditions in spatial indexes). Using this method, the context given above would instead become:

```
{
 { B.B(B_0) || B.B(B_2) } && Std.Compare(B.B1(), ">", 45);
}
```

This example condition is illustrated further in Figure 6.8 which also demonstrates the use of the "or" connective to specify more than one set of axioms which can be matched to fulfil the condition. In our example, we have stipulated that the return value for B.B can be either 0.0 or 2.0. The code has

```
{ Std.Compare(B.B2(), "==", 52.2) &&
 {
   Std.Compare(A.A3(), ">=", 102.9)
 }
}
```

(a) The Condition

```
ContextCond
 and (hasContextContract some
    (ContextRslt
     and (hasContext some B_Context)
     and (hasContract some B2_Contract)
     and (hasResult some
        (OWLList
         and (hasContents some
            (OWLList
             and (hasContents some
                (Parameter
                 and (hasType value "xsd:double"^^string)
                 and (hasValue value "52.0"^^double)))
             and (hasNext some EmptyList)))
         and (hasNext some EmptyList)))))
     and (hasContextContract some
        (ContextRslt
         and (hasContext some A_Context)
         and (hasContract some A3_Contract)
         and (hasResult some
            (OWLList
             and (hasContents some
                (OWLList
                 and (hasContents some
                    (Parameter
                     and (hasValue some double[>= "102.9"^^double])
                     and (hasType value "xsd:double"^^string)))
                 and (hasNext some EmptyList)))
             and (hasNext some EmptyList)))))
```

(b) The OWL Code

Figure 6.7: A Condition with a Nested Block

been sparsely commented using the # character to indicate the beginning of a line comment.

These transformations are summarised in table 6.5.

6.5.2.6 Representing ∃ Blocks

∀ blocks are first converted to a sequence of contracts connected with AND operators. They are then handled in the same way as already discussed.

∃ statements can be represented using a combination of AND and OR logical operators (as discussed in the previous chapter). However, this can result in extremely large and complicated conditions so we have chosen to handle them differently.

When ∃ blocks are parsed we produce a ContextCond class whose structure is quite different from that generated for normal conditions. This class requires that an ExistsResult axiom is present in the ontology with an associated hasName data property which specifies a unique string value.

In addition to the class generated above, we also generate an ExistsBlock class and ContextCond classes for each of the contracts within the ∃ block. The ExistsBlock class is linked to each of these ContextConds via a hasExistsStatement object property. These ContextCond classes are superclasses of the ExistsStatement class so that the ExistsStatement class can be used in its place for readability. We evaluate each ContextCond class within the block once prior to evaluation and add an ExistsResult individual with the appropriately named unique string value to the ontology if the number of passing contracts is within the min/max bounds specified in the ∃ opening statement. Pseudo code for this algorithm and examples of OWL code are available in Figures 6.9 and 6.10 respectively. Note that the code in these figures makes reference to an i_full_context individual; the structure of which is explained in the following section.

6.5.3 Representing Context Values

Context values are represented as instances of the ContextRslt class, consisting of a number of individuals connected by object and data properties. Figure 6.11 illustrates this structure.

Parameter instances are connected to the actual values given for the Context Values parameter list and result value.

We have attempted to reduce the size of the ontology by maintaining only single instances of a class where possible. For example, as discussed earlier there exists only one instance of each Context and ConcreteContext class and these are used by all ContextRslt instances.

We define a single ContextCond individual (named i_full_context) which is linked to each ContextRslt instance via the hasContextContract object property.

```
{ !B.B(B_1) && !Std.Compare(B.B1(), "<=", 45); }
```

(a) The Condition

```
ContextCond
 and (hasContextContract some # B.B(B_0) || B.B(B_2)
    (ContextRslt
     and (hasContext some B_Context) and (hasContract some B_Contract)
     and (((hasParameterList some (OWLList # B.B(B_0)
         and (hasContents some (OWLList
             and (hasContents some (Parameter
                 and (hasType value "xsd:string"^^string)
                 and (hasValue value "B_0"^^string)))
             and (hasNext some EmptyList)))
         and (hasNext some EmptyList)))
     and (hasResult some (OWLList
         and (hasContents some (OWLList
             and (hasContents some (Parameter
                 and (hasType value "xsd:double"^^string)
                 and (hasValue value "0.0"^^double)))
             and (hasNext some EmptyList)))
         and (hasNext some EmptyList))))
         or
         ((hasParameterList some (OWLList # B.B(B_2)
         and (hasContents some (OWLList
             and (hasContents some (Parameter
                 and (hasType value "xsd:string"^^string)
                 and (hasValue value "B_2"^^string)))
             and (hasNext some EmptyList)))
         and (hasNext some EmptyList)))
     and (hasResult some (OWLList
         and (hasContents some (OWLList
             and (hasContents some (Parameter
                 and (hasType value "xsd:double"^^string)
                 and (hasValue value "2.0"^^double)))
             and (hasNext some EmptyList)))
         and (hasNext some EmptyList)))))))
 and (hasContextContract some # Std.Compare(B.B1(), ">", 45)
    (ContextRslt
     and (hasContext some B_Context)
     and (hasContract some B1_Contract)
     and (hasResult some
        (OWLList
         and (hasContents some (OWLList
             and (hasContents some (Parameter
               and (hasValue some integer[> "45"^^integer])
               and (hasType value "xsd:integer"^^string)))
             and (hasNext some EmptyList)))
         and (hasNext some EmptyList)))))
```

(b) The OWL Code

Figure 6.8: A Condition with two Negated Contracts

```
function PREPROCESSEXISTS(existBlocks, i_full_context)
    for existBlock : existBlocks do
        numPasses ← 0            ▷ Count the number of successful conditions.

        min ← PARSEMINVALUE(existsBlock)
        max ← PARSEMAXVALUE(existsBlock)

        existStatements ←
                GETOBJPROPERTYRANGE(HAS_EXISTS_STATEMENT)

        ▷ Evaluate every condition associated with this ExistsBlock.
        for existStatement : existStatements do
            if PASSESEVAL(existStatement, i_full_context) then
                numPasses ← numPasses + 1
            end if
        end for

        if numPasses ≥ min and numPasses ≤ max then
            existBlockId ← GETID(existBlock)

            ▷ Create an ExistsResult individual with a hasName value property.
            existsResult ← CREATEINDIVIDUAL(EXISTS_RESULT,
                                            HAS_NAME, existBlockId)
            i_full_context ←ADDOBJPROPERTY(i_full_context,
                                           HAS_EXISTS,
                                           existsResult)
        end if
    end for
end function
```

Figure 6.9: The Algorithm for Preprocessing Exists Blocks

```
{ exists (1, 2)
  Std.Compare(B.B(), "<=" 100.5),
  A.A(A_1),
  Std.Compare(A.A(), "==", 25);
}
```

(a) An ∃ block with three conditions

```
ContextCond
 and (hasExists some
  (ExistsResult
   and (hasName value
     "c_msg_10_context_cond_0_1_Exists_block_1$1_3_result"^^string)))
```

(b) OWL code for the ∃ form of Contextual Condition

```
Class: c_msg_10_context_cond_0_1_Exists_block_1$1_3
ExistsBlock
 and (hasExistsStatement some c_msg_10_context_cond_0_1_Exists_statement_0)
 and (hasExistsStatement some c_msg_10_context_cond_0_1_Exists_statement_1)
 and (hasExistsStatement some c_msg_10_context_cond_0_1_Exists_statement_2)
```

(c) OWL representing the three conditions present in the ∃ block

```
individual: i_full_context
 Types: ContextCond

 Facts: hasExists indi_exists_1

individual: indi_exists_1
 Types: ExistsResult

 Facts: hasName c_msg_10_context_cond_0_1_Exists_block_1$1_3_result
```

(d) A potential i_full_context individual matching the condition in Figure 6.10b

Figure 6.10: Representing ∃ Conditions

Op	Value	New Value
Numeric Values		
Any	IntValue	IntValue^^xsd:integer
Any	FloatValue	FloatValue^^xsd:double
Any	DateValue	DateValue^^xsd:dateTime
=	StringValue	StringValue^^xsd:string
Boolean Values		
=	true	true^^xsd:boolean
=	false	false^^xsd:boolean
Ontology Values		
=	AFTERNOON (1.0)	1.0^^xsd:double

(a) Mapping to OWL Data Types

Op	Value	New Op	New Value
Numeric and String Values			
$\neq$	NumValue	$<$	NumValue^^xsd:dataType
and		$>$	NumValue^^xsd:dataType
$\not<$	NumValue	$\geq$	NumValue^^xsd:dataType
$\not\leq$	NumValue	$>$	NumValue^^xsd:dataType
$\not>$	NumValue	$\leq$	NumValue^^xsd:dataType
$\not\geq$	NumValue	$<$	NumValue^^xsd:dataType
Boolean Values			
$\neq$	true	=	false^^xsd:boolean
$\neq$	false	=	true^^xsd:boolean
Ontology Values			
$\neq$	AFTERNOON (1.0)	=	MORNING (0.0^^xsd:double)
or		=	EVENING (2.0^^xsd:double)

(b) Mapping Negated Contracts to OWL Representation

Table 6.5: Rules for Mapping values to their OWL Representation

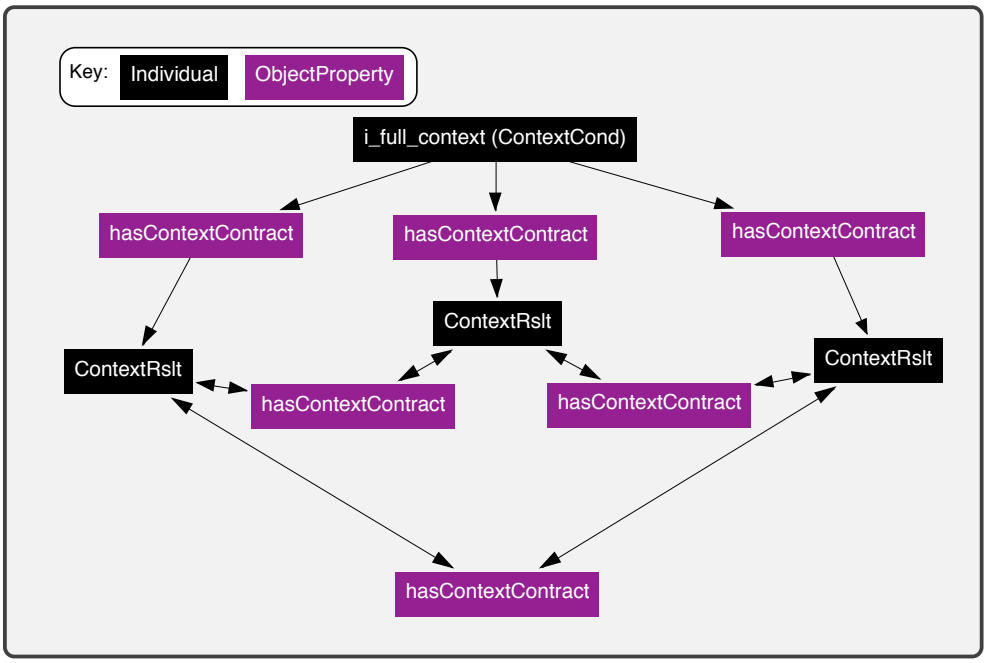


Figure 6.11: Representing All Observable Context Information

Prior to evaluation we calculate the Cartesian product of all ContextRslt individuals and link them using the hasContextContract object property, resulting in a fully connected subgraph of ContextRslt individuals. This is necessary to ensure the correct evaluation of nested blocks.

After these amendments have been made the i_full_context individual represents the complete observable context at a given time.

6.5.4 Evaluating Conditions

Conditions are evaluated by running the OWL reasoner to infer class members - i.e. to infer which individuals are instances of a class within the ontology. If the i_full_context individual is found to be a member of an ContextCond class then we can safely infer that the condition represented by this ContextCond class has succeeded and that the associated message may be received by the requesting participant. See Figure 6.12 for an example which demonstrates a case where the illustrated i_full_context individual in Subfigure 6.12c would be inferred as a member of the Condition illustrated in Subfigure 6.12b.

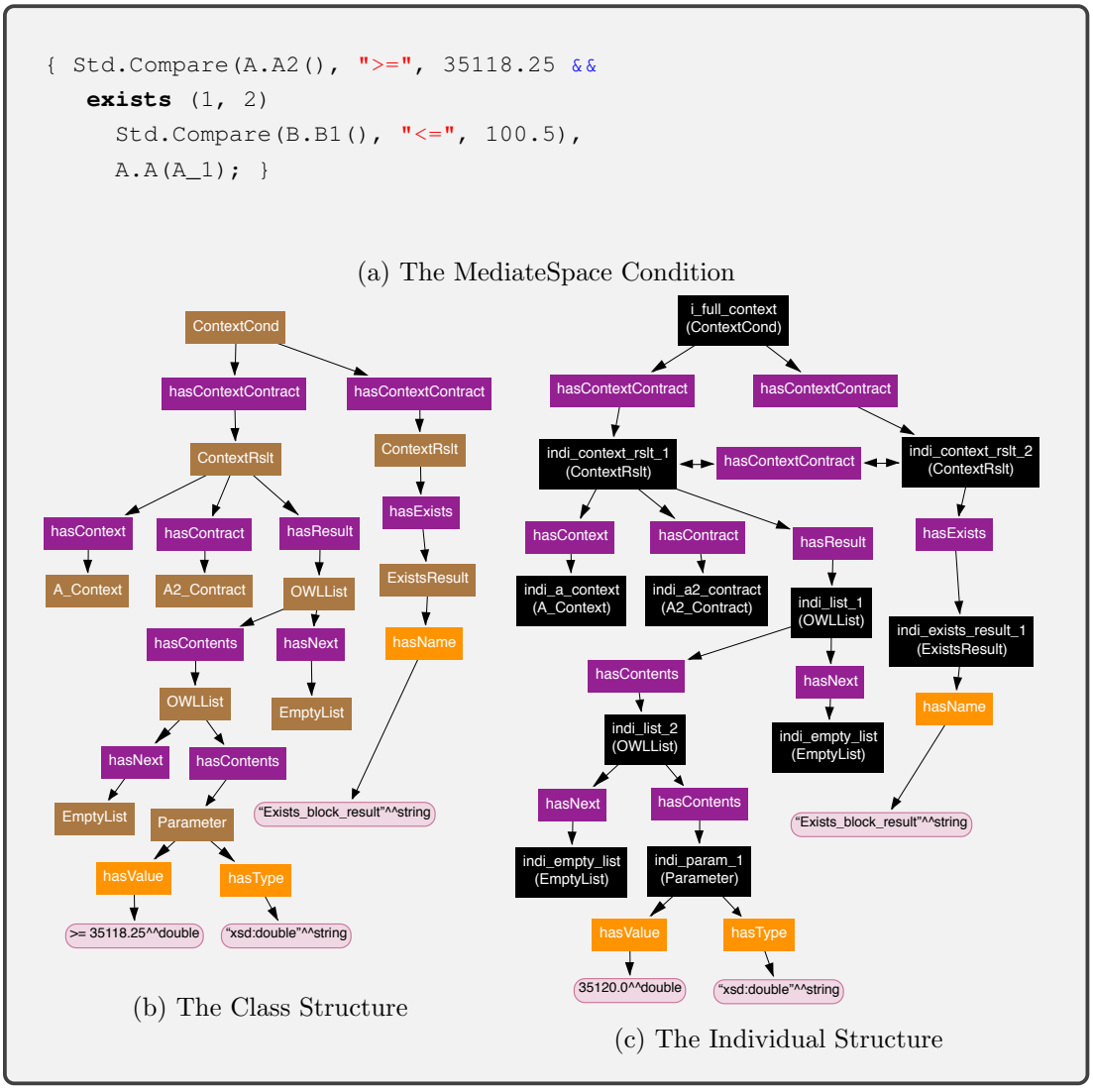


Figure 6.12: An Example Illustrating OWL code structure and Inference

6.6 Summary

In the preceding chapter we discussed our methodology for translating contextual conditions to an OWL representation for evaluation.

We have now discussed in detail each of the major components of our system; the middleware platform, the spatial indexing algorithm and the OWL representation. In the following chapter we provide an overview of our system design and describe the methodology followed for evaluating each element of our system.

7 Design and Experimental Setup

7.1 Introduction

This chapter discusses the design of the MediateSpace system and our experimental setup. We first provide an overview of our system design and then move onto our experimental setup, which is divided into three sections:

- Context Modelling
- Benchmarking
- Simulation

This first section discusses how we have modelled contextual conditions. Specifically, we consider condition complexity and our methods for controlling this complexity during experimentation. This model is used in both benchmarking and simulation.

The second section discusses our methodology for benchmarking the evaluation of contextual conditions. This is an important consideration, as if evaluation is inefficient it could become a major bottleneck which would impact throughout and user response times.

The final section concerns the design of our simulation, and provides an overview of the measured variables and the user modelling performed to provide additional realism to the simulation.

7.2 MediateSpace Design

This section provides an overview of our design and discusses major implementation decisions and any design patterns [36] employed.

7.2.1 Nodes

Nodes represent the physical devices in the network. There are two types of node: *Regional* and *Participant*. However, their basic structure is largely

Capability	Description
In/Out Queues	For storing incoming and outgoing messages. These queues are processed periodically.
NetworkComms Interface	For dispatching messages to other nodes within the network. This class handles all communication with the simulator so that Nodes do not need to be aware of the environment they are running in.
SpatialComms Interface	For interacting with the spatial indexing data structure. In our implementation this is an R-Tree.
Location Reader	For retrieving the current and next geographical location of the node.
Internal/External Tuple Spaces	For storing local tuples and triggering interaction with other nodes in the network.

Table 7.1: Node Capabilities

identical and our discussion will only distinguish between node type when necessary.

This subsection has the following structure:

1. We provide a summary of the properties and capabilities available to each node,
2. We describe the file system structure employed by each node,
3. We describe the design of the tuple spaces held by each node.

7.2.1.1 Node Properties and Capabilities

Each Node has a *unique ID*, a *geographical location* and a number of capabilities which are summarised in Table 7.1.

Tuples are inserted into the in or out queue, depending on whether they are entering or leaving the node respectively. These queues are processed periodically.

Each node also has a location reader which allows them to obtain their current location and to update their location. In our simulation node locations are calculated according to a number of probability distributions, discussed in Section 7.4.3.2.

The remainder of these capabilities are discussed in more detail in the following sections.

7.2.1.2 File System Structure

Each node has an associated directory on disk which we call their repository. Their repository is used to store the contents of all tuples belonging to the node in MediateSpace language form. The major benefit of this approach is that the tuples are stored persistently, meaning that this data will not be lost if a node crashes. Additionally, because the tuples are stored in MediateSpace language form, they can be loaded into any implementation of the system provided it is capable of parsing our language.

Each node is associated with the directory structure in Figure 7.1a. Note that each tuple type is stored in a separate directory which makes it straightforward to load a single type. If it becomes desirable to store tuples of different types together they may be stored within the "ms-application" directory.

The repository is also responsible for storing contract handlers which are Java classes that implement the IContractImplementation interface with the single method in Listing 7.1. The process of loading and executing these IContractImplementation classes is discussed in more detail in Section 7.2.6.2

Each class represents a particular Concrete Context and it is responsible for establishing which contract should be executed and querying the appropriate sensor with parameter values if available.

In addition, each node has an associated property file which specifies the factory classes to use when constructing a number of system objects. Thus, it is possible to change the types of object created during runtime without recompilation. The available properties are shown in Figure 7.1b and we refer back to it as appropriate during the remainder of this section.

7.2.2 Tuple Spaces

Our tuple spaces are implemented as wrapper classes around the LighTS tuple space, which is a lightweight tuple space implementation developed by Picco, Balzarotti and Paolo [66].

Tuple objects offer fields which are used to store useful information about the language structure being processed. For instance, we can store meta field values such as the source user Id, tuple name or any other information that can be represented as an Object such as its OWL representation. Traditionally, fields in a tuple are numbered and ordered using this numeric value. We chose to use the more convenient "named fields" provided by LighTS which allow us to refer to fields using a string identifier.

As discussed in Chapter 4, each node has an internal and external tuple space, and each tuple space has several properties. These properties are discussed in the following subsections.

```
nodeId
├── persistent-space
│   ├── concrete-context-space
│   ├── context-space
│   └── message-space
├── semi-persistent-space
│   ├── context-request-space
│   ├── context-value-space
│   ├── message-match-space
│   └── message-request-space
├── ms-applications
└── contract-code-base
```

(a) The Directory Structure for a node with id "nodeId"

```
mediateSpaceFactory=LightsMediateSpaceFactory
internalRepFactory=NodeInternalRepFactory
externalRepFactory=MSLExternalRepTupleFactory
reasoningImplFactory=ManNodeOntologyFactory
owlApiReasonerFactory=FaCTPlusPlusReasonerFactory
queryFactory=NodeNetworkQueryFactory
```

(b) Property File specifying Factory Classes (without package names)

Figure 7.1: Files and Directories Available to a Node

7.2.2.1 Sub-Spaces

Each tuple space contains several tuple sub-spaces which each contain certain types of tuple. The tuple sub-spaces supported within these internal and external spaces vary depending on the type of node.

In order to support the tuple sub-space functionality we have associated each tuple space with a Mediator object [36]. The Mediator is responsible for filtering tuples to their appropriate sub-space and for returning a reference to the appropriate sub-space given a tuple type when requested.

7.2.2.2 Addition and Removal Notification

When a tuple is added or removed from a tuple space, the node owning the space will be notified.

Each tuple space and subspace are observable [36], meaning that a class

can choose to observe a space and receive notifications whenever an action is performed on it (rd, in, out, rdp, inp, rdg, ing, outg). Tuple spaces observe each of their subspaces so when actions are performed on a subspace any notifications are propagated to their parent space, and then onto any objects which chose to observe the parent.

Each tuple space has an associated event handler which observes it and performs appropriate actions based on the tuple notifications it receives. The actions performed tend to depend on the origin of the tuple i.e. if the notification was caused by the creation or modification of a tuple locally then the action performed will usually be different than when the tuple was received from a remote node.

We chose an observer-based implementation because it affords us a straightforward way of notifying the appropriate tuple spaces and event handlers without requiring them to be explicitly coupled to the tuple space they are monitoring. It is also potentially much more efficient than polling the space for changes. If the space was polled too often it would result in a lot of wasted CPU cycles; whereas if it was polled too seldom it would likely lead to delays in tuples being received.

7.2.2.3 Remote Communication

Each tuple space (with the exception of the Participant internal space) can be used to distribute and receive tuples to and from tuples spaces held on remote nodes.

Each of the network-capable tuple spaces (both regional tuple spaces and the participant external space) have an event handler which implements the *TupleSpaceNetworkHandler* interface. This contains the single method in Listing 7.2 which should be called by remote nodes to insert tuples into the appropriate space. The tuple(s) will then be processed as appropriate. Thus, to allow remote nodes to access a tuple space you simply need to allow them to execute one method (possibly via Remote Method Invocation [44]).

We separated the event handler classes from the tuple space as this gives us additional flexibility in deciding how tuple events are handled. For example, although in our implementation each node carries its own event handler, we could instead situate event handlers on remote machines and make each one responsible for forwarding events to multiple nodes based on the destination field of the inserted tuple.

7.2.3 The NetworkComms Interface

The NetworkComms interface has three roles which will now be discussed in the following subsections.

```
getResult(String contractName, List<MSTypedListValue> params)
```

Listing (7.1) The IContractImplementation Interface

```
void tupleInsert(String tupleContent, boolean routeComplete);
```

Listing (7.2) The TupleSpaceNetworkHandler Interface

```
void offerTuple(String from, String to, String tupleContent);
void offerTupleNow(String from, String to, String tupleContent);

void dispatchTuples();

ILocationRef getLocation(TupleSpaceType type, int nodeId);
```

Listing (7.3) The NetworkComms Interface

```
void insertLocation(ILocationRef p, int myUniqueId, int numMsgs);
boolean deleteLocation(ILocationRef p, int myUniqueId);

int insertMessage(PolyRectangle p, int myUniqueId,
                  int msgUniqueId);
void removeMessage(PolyRectangle p, int msgId, int myUniqueId);
Set<MsgLookupInfo> lookupMessages(PolyRectangle poly);

List<RegionalNodeBundle> getClosestRegionalNodes(ILocationRef p,
                                                 int numNodes);
```

Listing (7.4) The SpatialComms Interface

```
String addTo(IOntologyChunk chunk, ETupleType tupleType,
             ERepositoryType stateType);

void removeFrom(String identifier, ERepositoryType stateType);
void removeFrom(ETupleType tupleType, ERepositoryType repoType);
void clear(ERepositoryType stateType);

boolean evaluate(String observableName, String queryName);
Set<OWLClass> getContextSubclasses(IOntologyChunk classId, boolean direct)

void setStateChanged();

InputStream getState(ERepositoryType repoType);
InputStream getChunk(ERepositoryType repoType, String chunkId);

void outputOntology(OWLOntologyFormat format, OWLOntologyStorer storer,
                    File saveTo);
```

Listing (7.5) The IOntologyReasoner Interface

Figure 7.2: System Interfaces

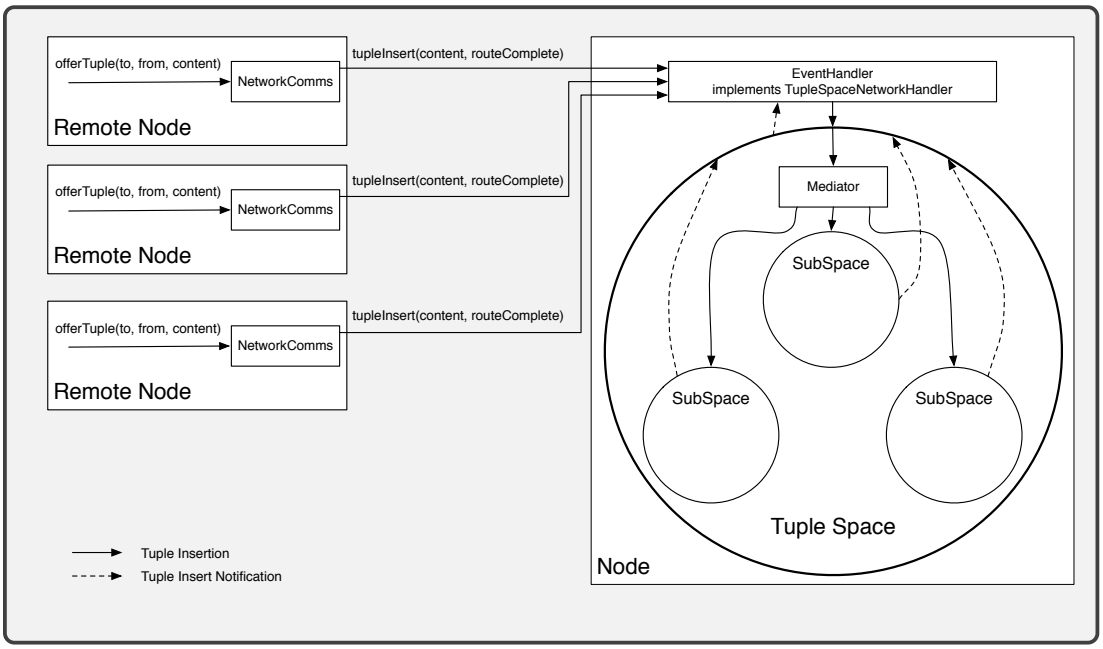


Figure 7.3: Tuple Space Operations and Network Handling

7.2.3.1 Simulator Communication

The NetworkComms interface allows us to communicate with the simulator in order to deliver messages to the appropriate node. This is desirable as it means that nodes can communicate with the NetworkComms interface and do not need an awareness of the simulator.

7.2.3.2 Message Queueing

Messages can be sent immediately to their destination by calling the offerTupleNow() method but by calling the offerTuple() method the message will be queued until the dispatchTuples() method is called. This latter option is useful as it allows us to send multiple messages without incurring the cost of establishing multiple connections. If a connectionless protocol is used either message queuing can be switched off or it can be used without modification as the queuing approach will likely not result in any negative consequences.

7.2.3.3 Location Lookup

The interface allows us to lookup the location of a node. When remote nodes bind to our local node we keep a record of their location; this method simply looks up that location in the local map.

7.2.4 The SpatialComms Interface

The SpatialComms interface provides access to the location and message spatial indexes, and has been designed to encapsulate these indexes so as to allow us to change the index used (e.g. exchanging the R-Tree for an X-Tree) without exposing this change to the user. The interface has four roles which will be discussed in the following subsections.

7.2.4.1 Inserting and Deleting Location

Regional nodes insert their location into the Location R-Tree on startup and remove it when they leave the network. If the Regional nodes are static they will insert their location only once. If they are non-static however they will reinsert their location at regular intervals.

7.2.4.2 Inserting and Removing Messages

Once a message has arrived at the appropriate Regional node it is inserted into the Message R-Tree so that other Regional nodes can perform message lookups and send requests for the message if appropriate. The message should be removed from the tree when the node leaves the network. If the node is non-static then all stored messages should be removed periodically and sent to a Regional node at a more appropriate location.

7.2.4.3 Obtaining Closest Regional Nodes

Both the Regional and Participant nodes need to lookup nearby Regional nodes to bind to. In the case of Participant nodes they need only lookup the closest Regional node as they bind to one node at a time. Regional nodes will attempt to find multiple nearby nodes to bind to. Regional nodes should refresh the list of nodes they are bound to periodically even if all Regional nodes are static. This is because new nodes may have entered the network since the last lookup was performed. Regional and Participant nodes change their location and attempt a rebind every 600 steps, which equates to 10 minutes in real-time.

7.2.5 Internal and External Representations

The internal representation refers to the data structure used to represent our MediateSpace language while the external representation refers to the String representation used when outputting language tuples into the network.

As can be seen in figure 7.1b the factory classes to use during the construction of internal and external representations are specified in a property file.

In our implementation we have chosen to represent our language tuples internally using a tree data structure and externally using the language specified in Chapter 4.

When the external representation (our language) enters a node's tuple space from a remote node it is parsed using the Java Compiler Compiler (JavaCC)[1] and a tree is generated using JJtree. JavaCC supports the automatic generation of parsers for languages defined using a BNF-like notation. JJTree is provided as part of the JavaCC software and supports the construction of trees which may then be traversed using the Visitor pattern [36].

The visitor pattern is then used in the construction of a number of object types within the system. Of particular note are the *TreeToMediateTuple* visitor class which translates the tree representation into Tuple object form and the ManOntology classes which translate our tree to Manchester OWL Syntax. As discussed above, useful properties of each language element are stored within the generated tuples using named fields.

The RepositoryLoader class has a number of convenience methods for parsing our language, creating tuples with the appropriate fields and returning said tuples for use. It can parse tuples from the file system repository discussed in Section 7.2.1.2 and also from strings, directories, files and streams. The RepositoryLoader class is also responsible for loading and instantiating contract drivers from the repository on the disk.

7.2.6 Tuple Space Services

A number of service classes are available for manipulating the tuples in the internal and external tuple spaces. These services are summarised in Table 7.2. The getContextValue() method of the ContextService class and the executeContract() method of the ReasonerService class are explained further in the following sections. We also present pseudo code for both methods which illustrate the use of the ReasonerService and ExternalRepresentationFactory classes, and demonstrate how tuple spaces can be queried through the use of templates.

7.2.6.1 Obtaining Context Values

The getContextValue() method has three main stages:

1. We attempt to obtain the requested context from the internal tuple space of the local Node.
2. If it is not available locally, we request the context from a bound Participant Node. This involves constructing an OWL class that represents the type of context information we require and using the constructed

[1] https://javacc.java.net/

class to query the reasoner for a list of Participants supporting the context.

3. We issue a request for the context to bound Regional Nodes if the local Node is not bound to any Participants that can satisfy the request. If multiple participants can service a request the geographically closest node is chosen.

A record of each Regional Node visited is stored in a field of the ContextRequest tuple to ensure that the same request cannot be made of a Regional Node more than once. Additionally, checks are performed to ensure that a request for this context information was not made in the recent past. This is to avoid wasting resources searching for it again when it is very likely that either a previous request is still being propagated through the network or the ContextValue is not available at this time.

The above process is carried out at each Regional Node in turn until either the ContextValue is found or the ContextRequest can no longer be propagated. This may be because the Node is bound only to Regional Nodes that have already been visited or the request has reached the maximum allowable distance from the originating Node. Pseudo code for this method is available in Figure 7.4.

7.2.6.2 Executing ConcreteContext Implementations

When a Regional Node issues a request for context information to one of its bound Participants it includes a CONCRETE_CONTEXT_NAME field which specifies the name of the ConcreteContext tuple to be queried for context information. When the request reaches the Participant Node the appropriate tuple is loaded from the internal tuple space and the fully qualified name of the class responsible for interfacing with the appropriate sensor on the local device is read from a field of the ConcreteContext tuple. This driver Class is instantiated using reflection and executed. Pseudo code for this process is available in Figure 7.5.

7.2.7 Loading and Operating the Reasoner

Each regional node has access to a single instance of an OWL reasoner and an ontology repository which is used to hold all of the owl ontology information currently loaded for the tuples in either tuple space.

The repository holds ontology information in Manchester syntax form and the repository is divided into Query (classes) and Observable (individuals) partitions to balance the amount of data held by any partition, thus potentially improving lookup and insertion speeds. Listing 7.5 summarises the IOntologyReasoner interface implemented by the reasoner. This interface allows clients to add “chunks” of an ontology to the repository, where a chunk

Service	Description
SpaceInteractionService	• Count the number of tuples of a type overall or in a specific space. • Perform a rd or in operation on all tuples matching a template. • Perform a rd or in operation on all tuples of a specific type. • Replace a tuple in the space by name and type.
ContextService	• Retrieve Context Value Tuples. • Buffer Context Requests i.e. keep a record of requests for context values and dispatch relevant values to these nodes if the enter the local node later. • Suppress repeated requests for Context Value Tuples.
ReasonerService	• Load the OWL representation for a specific type and partition. • Load the OWL representation for a specific named tuple. • Clear a partition of the repository.
ContractExecutionService	• Execute the driver program for a particular Concrete Context to obtain a Context Value Tuple.

Table 7.2: Summary of the Tuple Space Services

```
IMediateTuple getContextValue(IMediateTuple contextRqstTuple,
                              Contract contract) {
  /*
   * Create a template tuple to try and locate an appropriate Context Value
   * tuple in the internal space.
   */
  IMediateTuple template = new MediateTuple();
  template.addKeyValueField(TUPLE_TYPE, CONTEXT_VALUE);
  template.addKeyValueField(CONTEXT_NAME, contract.getContextName());
  template.addKeyValueField(CONTRACT_NAME, contract.getContractName());
  template.addKeyValueField(CONTRACT_PARAM_VALUES, contract.getParams());
  value = getInternalSpace().rdp(template);

  if (value != null) return value;

  // We haven't initiated a search for this contract in the recent past.
  if (!recentRqst.contains(contract)) {
    recentRqsts.add(contract);

    // Update the list of visited nodes.
    Set<String> visitedNodes = tuple.getFieldValue(VISITED_NODES);
    visitedNodes.add(this.getUserId());

    boolean success = requestFromParticipants(contextRqstTuple, contract);

    if (!success) {
      requestFromRegionals(contextRqstTuple, contract);
    }
  }

  return null;
}

boolean requestFromParticipants(IMediateTuple contextRqstTuple,
                                Contract contract) {
  /*
   * Load the OWL code for Contexts and ConcreteContexts
   * into the ontology repository.
   */
  reasonerService.loadContextTupleOWL(QUERY_PARTITION);
  reasonerService.loadConcreteContextTupleOWL(QUERY_PARTITION);

  /*
   * Construct a query to find ConcreteContexts for
   * the given contextRqstTuple.
   */
  availableConcretesQuery = OntConditionBuilder.and(
                                          contract.getContextName(),
                                          CONCRETE_CONTEXT_TUPLE);
  Set<OWLClass> availableConcretes =
                  reasoner.getContextSubclasses(availableConcretesQuery);

  if (availableConcretes.isEmpty()) return false;

  // The default behaviour is to choose randomly.
  OWLClass chosenConcrete = chooseConcreteContext(availableConcretes);

  IMediateTuple concreteContextTuple =
                reasonerService.getTupleForIRI(chosenConcrete.getIRI());

  IMediateTuple newRqstTuple =
                      externalRepFactory.createContextRequestTuple(
                                          concreteContextTuple,
                                          contract.getContextName(),
                                          contract.getContractName());
  internalSpace.out(newRqstTuple);

  return true;
}

void requestFromRegionals(IMediateTuple contextRqstTuple,
                          Contract contract) {
  IMediateTuple newRqstTuple =
                      externalRepFactory.createContextRequestTuple(
                                          contextRqstTuple,
                                          contract.getContextName(),
                                          contract.getContractName());
  externalSpace.out(newRqstTuple);
}
```

Figure 7.4: ContextService getContextValue() Method Pseudo Code

```
IMediateTuple executeContract(IMediateTuple contextRqstTuple,
                              Contract contract) {
  IContractImplementation driver = loadDriver(contextRqstTuple);

  /*
   * Execute the driver, get the result and create a Context Value
   * Tuple for dispatching to the requester.
   */
  List<MSTypedValue> result = driver.getResult(contract.getContractName,
                                               contract.getParamValues());

  return externalRepFactory.createContextValueTuple(contract, result);
}

IContractImplementation loadDriver(IMediateTuple contextRqstTuple) {
  String concreteContextName = tuple.getFieldValue(CONCRETE_CONTEXT_NAME);

  /*
   * Get the Concrete Context Tuple, discover the name of the driver
   * and load the driver class via reflection.
   */
  IMediateTuple template = new MediateTuple();
  template.addKeyValueField(TUPLE_TYPE, CONCRETE_CONTEXT);
  template.addKeyValueField(TUPLE_NAME, concreteContextName);
  IMediateTuple concreteContextTuple = internalSpace.rdp(template);

  String driverName = concreteContextTuple.getFieldValue(CONTEXT_DRIVER);

  return repoLoader.loadContractDriver(driverName);
}
```

Figure 7.5: ReasonerService executeContract() Method Pseudo Code

usually refers to the Manchester representation of a Tuple or a contextual condition. Chunks may be retrieved and removed by Id and can be removed by Tuple Type. The stateChanged() method allows clients to notify the reasoner of changes to the ontology so that the reasoner can re-evaluate if necessary.

To perform evaluation the ontology is loaded and manipulated using the OWL API [51] and evaluation is performed using an OWL reasoner.

The OWL API makes it straightforward to create, modify and output OWL ontologies using a variety of syntaxes (e.g. Manchester or Turtle). The API also provides an interface for interacting with OWL reasoners and includes a ReasonerFactory which allows the developer to replace the reasoner being used without requiring recompilation. In our case we are using FaCT++ [86], which is an efficient OWL Reasoner written in C++ and communicated with using JNI bindings.

7.3 Context Modelling

In order to perform benchmarks and simulations we need to generate a large number of contextual conditions of varying complexity. To this end we have written a condition generator which takes a number of parameters to allow

the complexity to be varied.

A significant number of parameters have been defined which influence condition generation. These are summarised in Tables 7.3 and 7.4. Parameters are defined as either:

Expected Values The number we feel best represents its value in real conditions created in the field.

Probabilities The probability that a condition will have the property defined. For example, the probability that two Contracts are connected with a logical conjunction or the probability that a quantification block is used instead of a logical block.

Each table is divided into two sections. The first section discusses parameters relevant to the construction of Context structures. Considering Context generation is important as these structures have a bearing on the conditions generated. The second section discusses those parameters involved in condition generation.

Each value in Table 7.3 is based on the example conditions given for our pervasive advertising and Geocaching applications and also on the definitions of context in Sections 3.1 and 3.4. The values given by default generate slightly more complicated conditions than our examples to ensure that the evaluation does not underestimate condition complexity. The remaining tables provide probabilities which we alter in various ways during evaluation to explore the properties of our system.

We now discuss each of these tables in more detail.

7.3.1 Expected Values

As discussed, each parameter in Table 7.3 specifies an expected value (unless specified otherwise). Note that our parameters distinguish between blocks and nested blocks as we believe they have different expected values. Specifically, we believe that there will be fewer nested blocks than top-level blocks and that nested blocks will contain fewer Contracts. We also support the Exists Divisor parameter which makes it straightforward for us to generate $\exists$ blocks with min/max parameters that are dependent on the number of Contracts within the block.

We use a binomial distribution to choose actual values during condition evaluation as this allows us to add an element of randomness to the generation process while still making it likely that the majority of conditions will be constructed using values around our expected value. The Binomial distribution is defined in Formula 7.1.

When a task with a measurable success or failure is performed n times, the binomial distribution allows us to determine the probability of succeeding in

this task k times. The probability of succeeding on each trial is given as p, and the probability of failure on each trial is q (p - 1).

$$P(n) = \binom{n}{k} p^k q^{n-k} \tag{7.1}$$

Binomial Distribution

One benefit of obtaining parameter values from a Binomial distribution is that its shape can be modified by applying different levels of skew. Figure 7.6 shows the Binomial distribution which represents an Expected Value of 3 with four levels of skew. Each of these examples show the result of generating 100,000 discrete random values from the binomial distribution with n = 6 and p = the chosen skew. The values of k range from zero to six.

A skew of 0.5 is specified as our base case. This produces a distribution which is roughly normally distributed. From Figure 7.6 we can see that as the skew gets more extreme on either side the expected value becomes much smaller (when the skew < 0.5) or much larger (when the skew > 0.5). We can leverage this property by altering the skew to produce more or less complex conditions.

To achieve the desired binomial distribution for each Expected value, we multiply it by two in order to place the Expected value in the centre of the distribution.

Each of the parameters defined in Table 7.3 may also be defined in terms of a maximum and minimum value, with the value being chosen uniformly between these bounds.

7.3.2 Probabilities

Probability-based parameters are represented as floating-point values in the range [0.0, 1.0], where the probability of each parameter grows as its value increases towards 1.0.

The HasConcreteContext parameter declares the probability of a given participant having a ConcreteContext structure for a given Context. For example, if this property has a value of 0.7 then each participant will have access to a ConcreteContext for approximately 70% of Contexts.

The HasMatchingContracts parameter gives the probability that a Contract will be specified more than once within a condition to constrain both sides of the Contract's dimension. For example, if we have the condition:

```
Std.Compare(A.A1(), ">", 20)
```

and the HasMatchingContracts parameter passes we append a matching condition which results in the condition:

```
Std.Compare(A.A1(), ">", 20) && Std.Compare(A.A1(), "<", 50)
```

Parameter	Description	Expected Value
Context Structure Generation Parameters		
NumContextTypes^{+}	The number of Context structures used	6
NumContractsPerContextType^{+}	The number of Contracts each Context should contain	3
NumContractParameters	The number of parameters a given Contract should require	2
NumOntologyConcepts	The number of ontology concepts supported by a given Context	4
Condition Generation Parameters		
NumBlocks	The number of top-level blocks a condition should have	2
NumContractsPerBlock	The number of Contracts (n) in each top-level block. If the SplitContracts flag is on, the Contracts are divided evenly amongst all top-level blocks rather than generating n separate Contracts for each block	3
NestedBlockDepth	The degree of nesting within a top-level block	1
NumContractsPerNestedBlock	The number of Contracts (n) to use within each nested block. If the SplitContracts flag is on, the n Contracts are divided evenly amongst all blocks nested at this level rather than generating n separate Contracts for each nested block	2
ExistsDivisor^{+}	If turned on via the ExistsDivisor flag, all ∃ blocks are of the form: ∃ (n/existsDivisor, n) where n = number of conditions within the block. When ExistsDivisor = 0 blocks instead take the form: ∃ (1, n)	2

$^{+}$ Has an absolute value, not an expected value. The value given is the default.

Table 7.3: Condition Generation Parameters with Expected Values

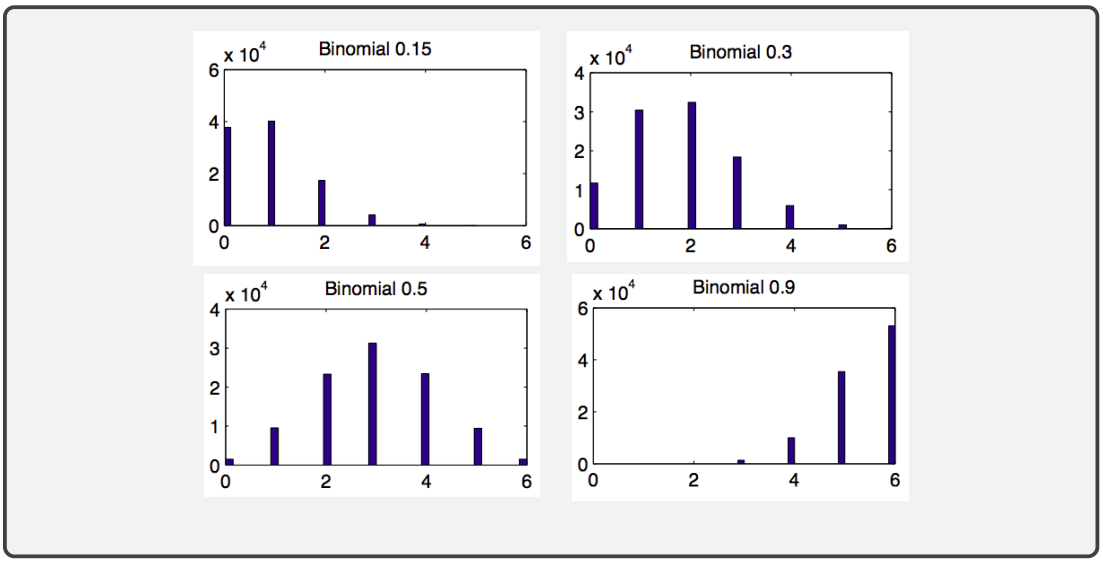


Figure 7.6: The Binomial Distribution with Different Levels of Skew

Finally, note should also be taken of the ContextValuePasses parameter which is responsible for determining the probability of generating a ContextValue structure which fulfils the constraint given in a contextual condition. For example, if a condition stipulates that:

```
Std.Compare(A.A1(), ">", 50) && Std.Compare(A.A1(), "<", 75) &&
Std.Compare(B.B1(), ">=", 25)
```

and the ContextValuePasses parameter has a value of 1.0, we can be certain that two ContextValues can be generated fulfilling the condition:

```
A.A1() == 55, B.B1() == 300
```

The HasConcreteContext and ContextValuePasses parameters are particularly relevant to simulation as they can impact the number of neighbours capable of obtaining context values and the number of succeeding conditions respectively. An increase in the number of succeeding conditions will result in a greater amount of activity in the network as more messages will be requested and received.

7.3.3 Message Request Condition Generation

When running simulations we need to generate message requests for participants to issue into the network. Message requests can be generated using all of the parameters discussed so far. However, for message requests we also wish to model how specific they are. That is, we wish to be able to specify how much of the message space is considered by a given request. For example,

Parameter	Description	Probability
Context Structure Generation Parameters		
HasConcreteContext	A given Participant will have a given ConcreteContext structure	1.0
Int/Float/Bool/Date ParamType	Choosing a Contract parameter data type	0.3/0.3/0.23/0.17
Condition Generation Parameters		
QuantificationBlocks	Generating a block using quantification rather than logical connectives	0.4
∀Blocks	Choosing a ∀ block rather than an ∃ block when quantification is chosen	0.8
Conjunction/Disjunction	Connecting two Contracts with a conjunction/disjunction	1.0/0.0
HasMatchingConditions	The same Contract is used multiple times to constrain the condition on both sides	0.7
ContextValuePasses	The probability that the ContextValue generated for a given condition will pass	1.0

Table 7.4: Condition Generation Parameters with Probabilities

if our system supports a total of four Contracts (divided evenly between two Contexts) we might have the two conditions:

```
{ Std.Compare(A.A1(), ">", 5) && Std.Compare(A.A1(), "<", 1000) &&
  Std.Compare(B.B1(), ">=", 200) &&
  Std.Compare(B.B1(), "<", 3000) && A.A(A_2) && B.B(B_1); }

{ Std.Compare(A.A1(), ">", 5) && Std.Compare(A.A1(), "<", 10) &&
  Std.Compare(B.B1(), ">=", 200) &&
  Std.Compare(B.B1(), "<", 259) && A.A(A_2) && B.B(B_1); }
```

The first condition will likely consider a much larger subset of the message space than the second. This first condition will likely result in the dispatch of a greater number of network messages. Thus, the specificity of requests is important to consider as it can impact the scalability of the network.

The specificity of requests is controlled by a number of parameters. Through the combination of the RangeSpecificity and PercentOfContracts parameters

Parameter	Description	Default Value
RangeSpecificity	How constrained the range should be for each Contract.	0.5
PercentOfContracts	The percentage of the available contracts to be included in the condition	0.5
Force&&Joins	**Logical Blocks**: All Contracts are connected with conjunctions. **Quantification Blocks**: All blocks must be ∀	false
ForceMatchingContracts	Each Contract must be used multiple times to constrain it on both sides	false
ForceSingleBlock	All Contracts specified within a single block	false
ContractsInEachBlock	Each block contains every contract from the set generated for the PercentofContracts parameter	false

Table 7.5: Message Request Parameters with Default Values

it is possible to broaden or reduce the volume of the hyper-rectangle created for a condition. Reducing the percentage of contracts included in each request necessarily broadens the hyper-rectangle as the dimensions representing any missing contracts will need to be fully unbounded. Increasing range specificity will reduce the bounds of the contracts supported. In our examples above, the first condition has very open bounds for the A.A1() and B.B1() Contracts; whereas the bounds of the second condition are much more restricted. The ForceMatchingConditions parameter can be used to restrict the index dimensions of each Contract further.

The remaining parameters allow you to increase specificity in a number of ways. Force&&Joins ensures that all of the Contracts within a block are connected using conjunctions, with the benefit being that this block will result in a single spatial index containing all available constraints. If disjunctions were permitted these constraints could be divided between multiple spatial indexes.

ForceSingleBlock ensures that all Contracts are within a single block. This can result in simpler conditions which require fewer spatial indexes to represent. If you wish to allow multiple blocks within a condition but still enforce specificity within each block the ContractsInEachBlock parameter can be used.

7.4 Simulation

We use the PlanetSim [3] network simulator to carry out simulations of our system. PlanetSim is a discrete-event simulator, meaning that simulations are measured using discrete steps independent of time. The number of steps to run depends on the measure of time a step represents and also depends on the application being simulated. For our application each step is modelled as a second in real time and we have chosen to run our simulations for a total of 43,200 steps each, which equates to 12 hours in real time.

A 12 hour simulation time is appropriate as it covers what we perceive to be the "active" hours of an average individual; leaving for work at approximately 8 AM and returning home at 8 PM or before. The remaining time is usually spent at home where a context-aware service is probably only used sparingly. The mapping of one step to a second seems appropriate as the amount of work each node performs on each step tends to equate to what we would intuitively expect a node to be able to process in one second.

PlanetSim has a tiered architecture, consisting of three layers:

Network Layer Responsible for modelling the behaviour of the underlying network.

Overlay Layer Provides an interface for implementing overlays which operate over the Network. The interface used is proposed in [26] and its use within PlanetSim is discussed in detail in [3].

Application Layer Represents the actual application which leverages the underlying overlay for network connectivity. The PlanetSim documentation includes an implementation of the SCRIBE event notification infrastructure [75] which sits atop the Pastry overlay.

Each layer contains callback methods which are called by the layer below when an event occurs. For example, the deliver(key, msg) method of the Application layer is called by its associated Overlay node whenever a message is received.

Our implementation only uses the Application layer, using the Network and Overlay implementations provided by the PlanetSim team. The overlay used for all simulations is Pastry [76]. Pastry is an appropriate choice because it takes the geographical proximity of nodes into account when building the routing table. Nodes in our middleware bind to nearby nodes as part of the protocol; thus the Pastry overlay will ensure that network messages reach their destination in a small number of hops.

A MediateSpace node (either Participant or Regional) is injected into each Application layer node within the simulated network, and any messages received are forwarded to our node via the tupleInsert(content, routeComplete) method discussed in Section 7.2.2.3.

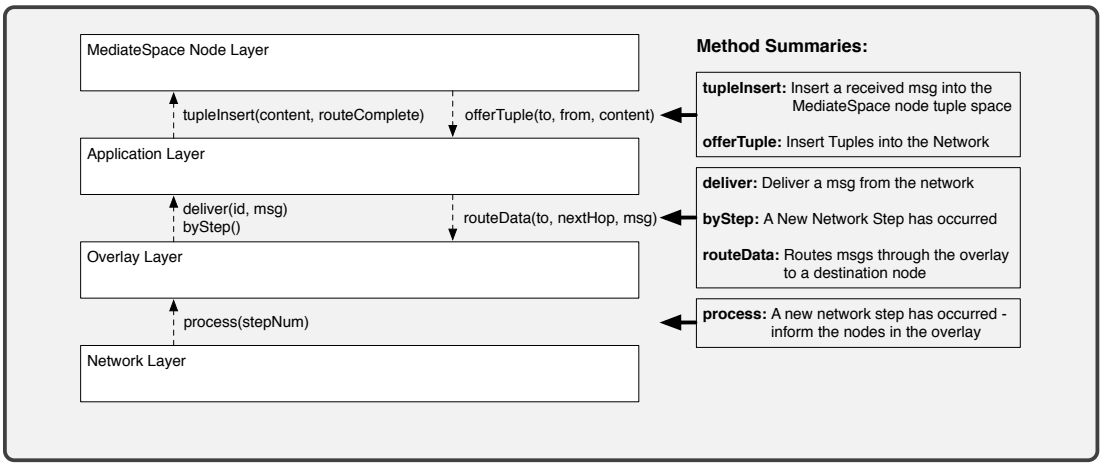


Figure 7.7: Communication between PlanetSim Layers and MediateSpace Nodes

PlanetSim represents node addresses within the network using NodeHandle objects. Our SimulationRunner class provides maps for converting between our numeric node id's and PlanetSim NodeHandles and vice versa.

Figure 7.7 summarises the PlanetSim architecture and how our MediateSpace application communicates with it.

There are a number of parameters that can be specified to modify the behaviour of the simulator. These are summarised in Table 7.6.

7.4.1 Achieving Reproducibility

The simulation process requires the generation of a large number of random values which are obtained from a variety of different random number generators. These range from the uniformly distributed values obtained when establishing a new node in the overlay to the Binomial values obtained during condition generation. In order to ensure that our simulations are reproducible we establish a master seed value which is specified as the seed of a uniform random generator. This random generator is then used to generate initial seeds for further random generators and so on until all generators have been seeded. This method ensures that our simulations are reproducible while only requiring us to specify a single master seed value.

7.4.2 Collecting and Processing Statistics

Statistics are recorded for every node in the network, with both "counter" and timing data collected. Timing data reflects the length of time in milliseconds that a specified task takes to complete whereas "counter" data represents the number of times a specified event occurs or type of message is received.

Parameter	Description
NumSteps	The number of steps to run the simulator for
InitialSeed	The seed used to initialise the random number generator
NumMsgs	The number of messages that each participant should insert into the network upon first joining
MaxRegionalNeighbours	The maximum number of Regional neighbours a Regional Node can have.
MaxParticipantNeighbours	The maximum number of Participant neighbours a Regional Node can have.
MaxContextDistance	The maximum geographical distance that a request for context information can travel from the originating node
Location Min/Max X/Y	The min and max X and Y values of the geographical space inhabited by nodes. These values represent the total area of the geographical space

(a) Basic Parameters

Parameter	Description	Default Value (secs)
AckPeriod	The period of time between the dispatch of acknowledgement messages into the network. By adding a slight delay several acknowedgements can often be combined to reduce network traffic	2
RetransmitDelay	The delay between a message being transmitted into the network and it being retransmitted if an acknowledgement has not been received	8
ContextValueExpiryTime	The amount of time a ContextValue tuple should remain valid	300
MsgMatchDispatchDelay	The delay between issuing requests for Context Values and performing message evaluation. This delay allows required ContextValue tuples to be received before proceeding with message evaluation	3

(b) Timing Parameters

Table 7.6: Simulator Parameters

Counter data is associated with the step on which the event occurred and this data can be aggregated, which gives flexibility over analysis.

Variables are represented in a tree structure, allowing higher level variables to base their collection partially or completely on an aggregate of the values held by one or more lower level variables.

We also divide all counter collections by origin of message. Specifically, each variable distinguishes between messages created locally and messages created by an external node.

Figure 7.8 provides a complete list of the variables considered, along with their structure. We are measuring the time it takes to receive both first and last replies for requests where appropriate because this will allow us to measure both the length of time it takes before a user receives any feedback for a request and the length of time until the request has been completely fulfilled.

7.4.3 Experimental Models

The modelling of behaviour is important to governments and industry as it allows them to make predictions about future events and behaviours. For example, weather forecasting is achieved through the modelling of weather patterns and insurance companies use modelling to calculate the premiums customers have to pay. These experimental models are often based on probability distributions which have been found to reflect the empirical data available for the area discussed. To ensure that our simulator is as realistic as possible we model the following:

- Human mobility
- Timings of Message Requests

We briefly discuss the Poisson distribution which until recently was used to model many types of behaviour. We discuss the assumptions which underpin the model and discuss why it should not be used as a model within our simulation. We then proceed to discuss alternatives for each of the required models.

7.4.3.1 Problems with the Poisson Distribution

The Poisson distribution allows us to determine the probability of a particular event occurring k times within a given time-frame provided we have an expected frequency of events within this time-frame. Although useful to model a number of physical phenomena this distribution makes a number of assumptions which render it inaccurate in many cases; namely it assumes the properties of “independent and stationary increments”.

The independent increments property stipulates that events are independent. That is, the probability of an event occurring at any given time cannot be influenced by past events. The number of children born each day worldwide

- NumTuples ..... Num Tuples Created and Num Received
  - NumContextTuples
  - NumConcreteContextTuples
  - NumContextRqstTuples
  - NumContextValuesTuples
  - NumMsgTuples
  - NumMsgMatchTuples
  - NumMsgRqstTuples
  - NumTuplePackages
  - NumSignalTuples
  - NumBindTuples
    - NumBindRejectTuples ..... Rejecting Bind as Already Bound to Max Number
      - NumBindRejectParticipants ..... Rejections to/from Participants
      - NumBindRejectRegionals ..... Rejections to/from Regionals
    - NumBindRequest ..... Num Bind Requests Created/Received
      - NumBindRequestParticipants
      - NumBindRequestRegionals
    - NumBindReply ..... Signifies a Successful Bind Attempt
      - NumBindReplyParticipants
      - NumBindReplyRegionals
  - NumUnbindTuples
- RetransmittedMsgs ..... Msgs have been Dropped and Retransmitted from this Node
- DuplicateMsgs ..... Number of Duplicate Msgs Received
- NumAcks ..... Number of Msg Receipt Acknowledgements Received

(a) Counter Variables

- MsgLookup ..... Evaluating Msg Rqsts (OWL Reasoner)
- ConcreteContextMatch ... Details of Participants Supporting a Context (Section 7.2.6.1)
- ContextRqstResponse ..... Waiting for a Response to a Context Rqst
- MsgRqstFirstResponse ..... Waiting for a Response to a Msg Rqst
- MsgRqstLastResponse ..... Waiting for the Last Response to a Msg Rqst
- MsgRqstFirstMsgReceipt ..... Waiting for the First set of Msgs for a Msg Rqst
- MsgRqstLastMsgReceipt ..... Waiting for the Last set of Msgs for a Msg Rqst

(b) Timing Variables

Figure 7.8: The Variables Considered within our simulation

does not satisfy this property as the probability of child birth will be affected whenever birth occurs.

The stationary increments property stipulates that the number of events which occur in any time interval should depend only on the length of said time interval. This assumes that the rate of event occurrence does not change. The number of customers entering a shop each hour would probably not satisfy this property as there is often a busy period such as lunchtime.

7.4.3.2 Modelling Human Mobility

The MediateSpace system is used to lookup and retrieve messages based on the user's current context, and it is anticipated that our system will be installed on mobile devices such as mobile phones and tablet computers. Thus, in order to ensure valid simulation results we must model user mobility and represent user location within the simulation. We now discuss a number of models which we have applied to our simulations. Each of the discussed models is expressed precisely below:

$$P(x) = \frac{e^{-\lambda}\lambda^{x}}{x!} \tag{7.2}$$

Poisson Distribution

$$P(x) \propto \frac{1}{x^{\alpha}} \tag{7.3}$$

Zipf Distribution

$$P(r_g) = \left(r_g + r_g^0\right)^{-\beta_r} exp\left(-r_g/K\right) \tag{7.4}$$

Barabasi's Radius of Gyration

$$P(\mathcal{T}) = \mathcal{T}^{-\alpha} \tag{7.5}$$

Power Law

Research by Barabasi [6] has found that many behaviours are better modelled as "bursty", with a flurry of activity followed by a long period of no activity. Among the behaviours seen to exhibit this distribution are E-Mail sending, web browsing, phone calls, human and animal sleeping patterns and even Darwinian Evolution [7]. This suggests that the Poisson distribution is inadequate when modelling these kinds of behaviours, and that it can be much more usefully modelled as a power law, which accounts for the outliers resulting from this bursty behaviour. Barabasi [6] suggests that this behaviour may be explained by an innate tendency to prioritise our activities into a priority queue, which will result in most of our activities being completed quickly - but with outliers caused by low priority activities which stay within our internal queue for extended periods.

Barabasi et al. [42] discovered that patterns of human mobility also follow a power law, with the vast majority of individuals restricting their movements

to a small area and only a few travelling large distances on a regular basis. Interestingly, it was found that this difference in mobility had little effect on the predictability of their movements. More specifically, they found that each individual has a representative radius of gyration which dictates the distances they travel, and that the probability of an individual having a certain radius follows the truncated power law in Formula 7.4 with parameters $r_g^0 = 5.8$ km, $\beta_r = 1.65 \pm 0.15$ and K = 350 km.

Barabasi et al. also discovered that people tended to frequent between 5 and 50 unique locations and that when ranked by number of visits the probability of finding an individual at a location L can be approximated as $P(L) \sim 1/L$ independent of the number of locations. Interestingly, this means that with high probability we will find any individual at one of their top-two locations 40% of the time. This property can be modelled using a zipf distribution (illustrated in Formula 7.3) with parameter $\alpha = 1$.

Wang, Han and Wang [88] found that people tend to stay at locations for quite long periods, reporting a staying time distribution which follows a power law with $\alpha = 1.98$.

These observations have been applied to our mobility model in the following ways:

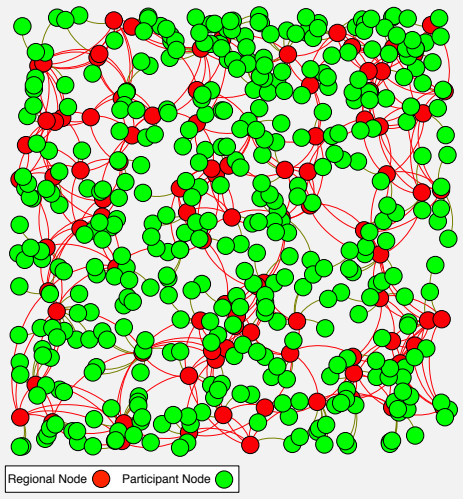
- Each user within the simulation is allocated a radius of gyration (G) taken from the distribution given in Formula 7.4.
- Each user is also randomly allocated a number of unique locations (taken from the set { 5, 10, 30, 50 }). The first location is generated by moving G units in a random direction within the two-dimensional location grid. The remaining locations are generated in the same way, always starting at the previously found new location.
- Each time a user changes position they obtain a staying time in seconds from the power law distribution in Formula 7.5 (with $\alpha = 1.98$).

However, we also wish to distribute users realistically within the geographical space. Research by Newman [65] has found city sizes follow a zipf distribution (Formula 7.3 with parameters $\alpha = 2.30$ and $xmin = 40000$), so we elected to calculate the bounds for 50 cities on our geographical grid and to distribute users to these cities in proportion to city size i.e. the larger the city, the more people present within it. Each user is given a random location within their chosen city.

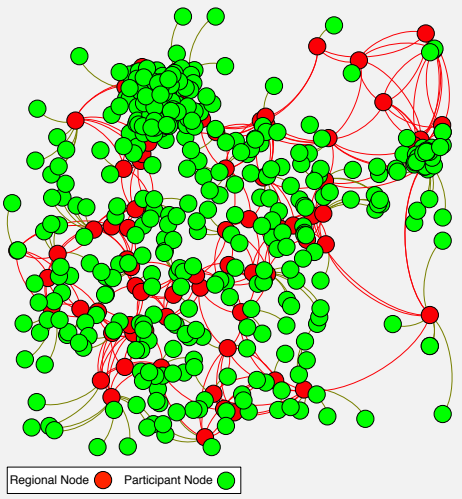
Subfigures 7.9b and 7.9c show an example of a network generated using the above models. Subfigure 7.9a shows a network generated using a uniform distribution for comparison.

We use the powerlaws[2] and zipf[3] libraries to generate probability distributions for use during the construction of user models.

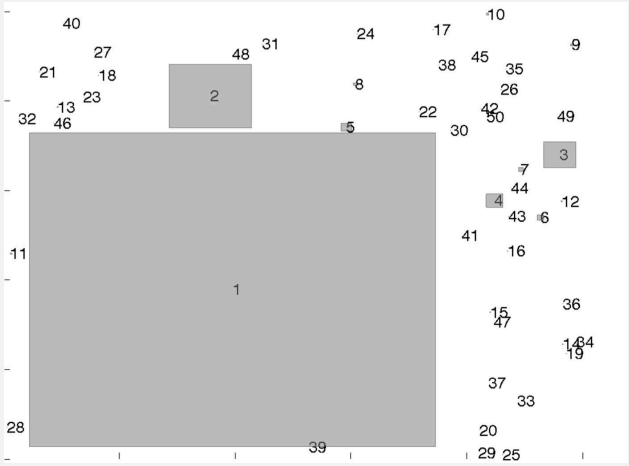
[2] https://github.com/Data2Semantics/powerlaws
[3] http://diveintodata.org/2009/09/13/zipf-distribution-generator-in-java/



(a) 500 Node, Uniformly Distributed



(b) 500 Node, Zipf Distributed (50 cities)



(c) Visualising the generated cities of Subfigure 7.9b

Figure 7.9: Examples of Simulated Networks

7.4.3.3 Modelling Network Request Timings

Our system allows users to issue requests for messages into the network. The number of requests and the delay between these requests must be modelled accurately to ensure that the results of any evaluation are valid. To model this behaviour we return again to the work of Barabasi [6] who documented the "burstiness" of many aspects of human activity. This pattern of regular activity interspersed with periods of inactivity has been found to hold true for a variety of message sending behaviours including E-Mail, instant messaging, job submissions on a supercomputer and FTP requests. We believe that valid comparisons between these activities and the task of issuing message requests into our network can be made so have chosen to model message requests as bursty in nature.

Message request times for each user are derived from the power law probability distribution specified in Formula 7.5 (with parameter $\alpha = 2$). After issuing a request each user retrieves a delay in seconds from the distribution. The user will not perform any further requests until this delay has passed. Users will continue this pattern until the maximum number of simulation steps has been reached. Thus, the number of requests per user will vary depending on the lengths of delay each user encounters.

7.4.3.4 Alternative Approaches

An alternative approach to mobility modelling could be to use data traces such as those provided by the MIT Reality data set [31]. These sets provide mobility data taken periodically from the mobile devices of real users. The main advantage of this approach is that provided the data was collected correctly we can be sure that the model is accurate. In addition, the model provides a history of user movement which is necessary when evaluating certain types of algorithm such as those used in pocket-switch networks. The disadvantages of this approach are that they limit the number of nodes to the number of individuals surveyed and that the data sets are finite, and thus can only be used to model relatively short periods of time. Although a trace driven approach could have been used to conduct our simulations, we chose a mathematical model because it gives us the flexibility to run additional simulations of arbitrary length, it does not restrict the number of nodes and our algorithm does not depend on movement history.

7.5 Benchmarking Condition Evaluation

Our contextual conditions are evaluated using the FaCT++ OWL reasoner. In order to assess the efficiency of this reasoning process we have used the

JMH benchmarking framework[4] which is created and maintained by the team responsible for the Java Just-in-Time (JIT) compiler. JMH makes it straightforward to benchmark code by providing a number of annotations to be applied to the fields and methods used during the benchmarking process.

In this section we first discuss the benchmarking process and the many issues which need to be dealt with to obtain valid results. We then discuss the specifics of what we have chosen to benchmark and discuss the parameters we provide to each invocation.

7.5.1 The Benchmarking Process

Obtaining valid benchmarks can be a complicated process, particularly when dealing with an optimising JIT virtual machine such as the JavaVM. Execution will always be slower at the beginning because the processor caches need to be populated and the JavaVM needs to complete its first stage of profiling in order to apply optimisations to sections of commonly executed code. JMH eliminates the effects of a slow start by executing the benchmarked code a number of times before starting to record execution times. These are the warmup iterations and the number to run can be set via command line or within the benchmarking class.

Once the warmup iterations have been completed the benchmark is executed a defined number of times and a summary of these iteration times is given. Possible summary information includes the throughput (number of times a method can be executed in a time period) and an average runtime. By performing a number of iterations JMH can obtain a more realistic indication of performance because the final result can account for the inevitable slight variance in runtime. It can also help to negate any unrepresentative iteration times caused by an uncontrolled variable such as an operating system maintenance cycle; although if at all possible these variables should all be controlled.

The above process is duplicated a number of times within different forks of the JavaVM. By executing sets of iterations in different forks we can obtain a significant number of timing data points without profiler problems.

We now move on to discuss what we have chosen to benchmark and how we overcame the potential pitfalls inherent in benchmarking.

7.5.2 Benchmarking the Reasoner

In order to assess the efficiency of condition evaluation we have run two types of benchmark:

- Ontology Loading and Reasoner Initialisation
- Message Evaluation

[4] http://openjdk.java.net/projects/code-tools/jmh/

We originally ran benchmarks which first loaded and initialised the ontology and then performed evaluation. However, we noted that the time it took to load our ontology into the reasoner was often considerable and may have a significant impact on our evaluation times. Thus, we decided to split the task into two separate benchmarks.

Before we discuss the specific details of our benchmarks we shall discuss the parameters of our code and the benchmarking framework.

7.5.2.1 Benchmark Parameters

Both benchmarks had a number of parameters which were specified at the command line. The JMH framework also accepts a number of parameters. These parameters are summarised in Table 7.7.

Each time a benchmark is run the two most important parameters are *num-msgs* and *data-index*. num-msgs indicates the number of messages to be loaded/evaluated by the reasoner during benchmarking, whereas data-index represents the class of data to be used. Data classes refer to a list of context generation parameters which define the complexity of the conditions the reasoner will be operating on. Context modelling and the context generation parameters are discussed in detail in Section 7.3. For example, we have a class of conditions labelled "expected-0.5" which consists of conditions generated from expected values with a skew of 0.5.

The JMH framework parameters specify the number of forks, number of warmup iterations and number of actual iterations. JMH also allows the specification of a number of profilers which output statistics about some facet of execution after each iteration. We chose to use garbage collection, JVM runtime and Java stack profilers to give us additional information about the execution.

7.5.2.2 Potential Pitfalls

The Java compiler (javac) performs optimisations during compilation. Ordinarily these optimisations are very welcome but they can cause problems when benchmarking [68]. For instance, javac will remove redundant code (i.e. code that is never executed or in the case of methods whose return values are never used). When benchmarking, classes and methods are often executed in isolation from the rest of their application as we wish only to benchmark a specific behaviour. This can result in method calls having no side effects outside of the method itself and having unused return values.

To ensure that the compiler does not optimise out these method calls we must ensure that the return value is always used and that the compiler cannot deduce the return value and replace the call with it at compile time. If a method is called n times within a loop, where n is decided at run-time (as in our code) a straightforward way of ensuring that this optimisation does not

Parameter	Meaning
msg-space	Disk location where the msg tuples to be evaluated can be found
context-space	Disk location where the Context tuples used within msg conditions can be found
context-value-space	Disk location where the Context Value tuples for each msg can be found
num-msgs	The number of msgs that should be loaded/evaluated
data-index	The index representing the class of data to generate

(a) Benchmark Code Parameters

Parameter	Meaning	Value
forks	The number of forks to perform	10
warmups	The number of warmup iterations to perform per fork	15
iterations	The number of actual iterations to perform per fork	20

(b) JMH Framework Parameters

Parameter	Meaning
gc, hs_gc	Standard and implementation specific garbage collection profiling
hs_rt	Implementation specific runtime profiling
stack	naive Java stack profiler

(c) JMH Framework Profiler Parameters

Table 7.7: Benchmark Code and JMH Framework Command Line Parameters

occur is to modify a variable with each call to the method. For instance, if the return type is numeric, the return value could be added to the variable on each return. In our case, the benchmark methods return booleans and we perform a logical disjunction on our variable upon each return.

Care must also be taken to ensure that different benchmarks are executed within different instances of the JVM. This is because the JIT compiler optimises the code during runtime to perform as efficiently as possible based on the history of execution. Thus, if several benchmarks are run in sequence the JVM may have already optimised for a previous benchmark and the results for the new benchmark may be unrepresentative.

7.5.3 Experimental Setup

All benchmarks were run on the University of Sussex HPC Bright Cluster using the Univa Grid Engine (UGE) batch system. The cluster runs Scientific Linux release 6.4. All jobs were submitted to the queue for a 12 core node containing two Intel X5650 processors with approximately 48 GB of RAM. Although the benchmarking software is single-threaded, we reserved all 12 cores to ensure that our benchmarking results could not be affected by the execution of other jobs.

The benchmarks were run on the OpenJDK Virtual Machine (version 1.7.0.51 x86_64) with 30 GB of heap space allocated at startup and 1 GB of stack space. We allocated a 30 GB heap at startup to ensure that the benchmarks would not be affected by heap resizing operations during execution. A 1 GB stack is allocated because our system can produce very large trees to represent complex conditions, and these need to be traversed.

Version 1.6.2 of the FaCT++ Reasoner is used, with the JNI bindings compiled locally for the Linux Operating System.

7.5.3.1 Common initialisation

Both benchmarks perform the same initialisation prior to each invocation of our benchmark method. This involves re-initialising the tuple spaces and reasoner to ensure that each inovation starts with a clean slate. In addition, all of the generated Context tuples and num-msgs messages are loaded into their tuple space.

A ContextValue tuple is generated for each Contract within each condition so, depending on the complexity of the conditions this could result in the generation of a large number of ContextValues. We load a single ContextValue for each possible Contract and choose these ContextValues randomly to ensure that message evaluation is as representative of real use as possible.

7.5.3.2 Ontology Loading and Reasoner Initialisation

This benchmark measures the amount of time it takes to perform all of the following tasks:

- Load the Manchester OWL code for every tuple into the reasoner,
- Build and insert into the ontology the i_full_context individual,
- Preprocess the ∃ statements.

The i_full_context individual represents the full contextual state at the time of creation. That is, it represents all of the ContextValue tuples in the space. This individual and the preprocessing of ∃ statements is discussed in detail in Sections 6.5.3 and 6.5.2.6 of Chapter 6.

7.5.3.3 Message Evaluation

As discussed above, we observed that reasoner initialisation times tend to be considerable so any message evaluation results which are not measured independently of initialisation tend to be masked by initialisation times.

Thus, in order to accurately measure execution times, in addition to the common initialisation discussed in Section 7.5.3.1 we initialise the reasoner once at the beginning of the benchmarking process and cache the resulting ontology so that it can be used for all future iterations without needing to reprocess it. This ontology is loaded into the reasoner prior to each invocation to ensure that we are only measuring message evaluation.

Message evaluation is performed simply by obtaining a list of all Messages in the tuple space and evaluating each one in turn. The benchmark ends when all messages have been evaluated.

7.6 Summary

In the preceding chapter we first provided an overview of the implementation of our MediateSpace system, which included a discussion of the interfaces we have defined for communication with the network. We then explained our methodology for modelling contextual conditions and defined the parameters used to control the complexity of these conditions. The third section concerned the simulation of our system, summarising the PlanetSim simulator, describing our method of gathering statistics and discussing the models of human mobility and network activity we used to make our simulation more realistic. Finally we discussed our benchmarking methodology, including a description of the JMH benchmarking framework, some potential pitfalls of benchmarking and our specific setups for the benchmarking of reasoner initialisation and condition evaluation.

The following chapter presents the results of our evaluation, offers some recommendations for optimal use of the system and considers several potential improvements.

8 Results

8.1 Introduction

This Chapter presents the results of our system evaluation. Specifically, we have evaluated the system along three dimensions:

Network analysis through simulation We used the PlanetSim network simulator to analyse patterns of message exchange and response times.

Contextual Condition evaluation times We have measured the execution times for a number of different classes of contextual condition using the JMH benchmarking framework.

Properties of the spatial indexing algorithm We have generated spatial indexes for a number of different classes of condition and have discussed how these classes affect the number of generated indexes.

We begin by describing the condition classes we have defined for evaluation and then proceed to discuss our system in terms of each of the dimensions listed above.

8.2 Condition Classes

We defined a number of condition classes that each exhibit an aspect of the contextual language that we wished to evaluate. We now briefly describe each of these classes in the following subsections.

8.2.1 Expected

This class represents the conditions we expect to be representative of those used in real deployments. All of the "expected" parameters in Table 7.3 are used with this class. Each parameter is fed into a binomial distribution in the manner described in Section 7.3.1. The resulting distributions are

used to select the actual values used during data generation. The use of a binomial distribution allows us to make random choices for each parameter while ensuring that a given value is the most likely to be chosen. We provide a "skew" parameter in the range [0.0, 1.0] that allows us to shift the distribution to make either larger or smaller values more likely. This is discussed in more detail in Section 7.3.1.

8.2.2 Expected Restricted

This class is similar to the above but with the difference that each Contract will have zero parameters and ∃ blocks are not permitted. This allows us to very significantly reduce the number of ContextValue tuples required to evaluate a condition and to eliminate the ∃ preprocessing step during reasoner initialisation respectively. The purpose of this class is explained further in Section 8.3.1.4.

8.2.3 Num-Conds-And and Num-Conds-Or

These classes restrict conditions to using conjunctions or disjunctions respectively, allowing us to illustrate the different effects that these operators have on our system. For instance, when a Message is processed for insertion into the Message tree the use of disjunctions will result in the generation of additional spatial indexes whereas the conjunction will not.

8.2.4 Exists-N-Div-2-N and Exists-1-To-N

The Exists-N-Div-2-N class generates conditions consisting only of ∃ blocks with an ExistsDivisor of 2. This generates blocks of the form: ∃ **(n/2, n)** where n represents the total number of Contracts in the block. Exists-1-To-N also generates only ∃ conditions but has an ExistsDivisor of 0. This is a special case and results in each block having the form: ∃ **(1, n)**. These classes can be used to establish the effects of different Min and Max parameters to the ∃ statement.

The classes are summarised in Table 8.1 using the parameters defined in Section 7.3 and examples from the generated test data for each class are given in Figure 8.1.

These classes were used extensively during the evaluation process.

8.3 Condition Evaluation Benchmarks

As discussed in Chapter 7 we performed benchmarks on our contextual conditions using the JMH benchmarking framework. 15 warmup iterations and 20 actual iterations were run for each benchmark, and this process was repeated

Set	Parameters	Value
expected	ExpectedSkew	{0.15, 0.3, 0.45, 0.5, 0.6, 0.75, 0.9}
expected-restricted	NumContractParameters	0
	∀Blocks	1.0
	ExpectedSkew	{0.15, 0.3, 0.45, 0.5, 0.6, 0.75, 0.9}
num-conds-and	NumBlocks	1
	NumContractsPerBlock	{1, 2, 4, 8, 16}
	Conjunction/Disjunction	1.0/0.0
num-conds-or	NumBlocks	1
	NumContractsPerBlock	{1, 2, 4, 8, 16}
	Conjunction/Disjunction	0.0/1.0
num-contexts	NumContextTypes	{4, 8, 16, 32}
	NumContractsPerContextType	{4, 8, 16, 32}
exists-n-div-2-n	NumBlocks	1
	QuantificationBlocks	1.0
	∀Blocks	0.0
	NumContractsPerBlock	{1, 2, 4, 8, 16}
	ExistsDivisor	2
exists-1-to-n	NumBlocks	1
	QuantificationBlocks	1.0
	∀Blocks	0.0
	NumContractsPerBlock	{1, 2, 4, 8, 16}
	ExistsDivisor	0

Table 8.1: Condition Classes

```
{ Std.Compare(platycoria.dolomedes(false), "==",83378.016); }
```

Listing (8.1) Expected, Skew = 0.15

```
{
  Std.Compare(extortioner.makhzan(-49,8.7,-27,1.7), "<",63.4)
  &&
  Std.Compare(extortioner.makhzan(-49,8.7,-27,1.7), "<",8.83)
  &&
  Std.Compare(newspaperese.organosol(8.74,1.9,true), ">=",27.88)
  &&
  {
    Std.Compare(extortioner.makhzan(-49,8.7,-27,1.7), ">=",38.2);
  };
}
```

Listing (8.2) Expected, Skew = 0.5

```
{
  Std.Compare(newspaperese.decernment(), "==",true)
  &&
  Std.Compare(precentorial.wungee(), "==",true)
  &&
  Std.Compare(inthronization.antitabloid(), "==",false)
  &&
  unnewness.unnewness(unnewness_1);
}
```

Listing (8.3) Expected Restricted, Skew = 0.5

```
{
  Std.Compare(unnewness.alfonso(2.8,true,10/02/2051), "<",18.25)
  &&
  Std.Compare(unnewness.syriac(-15,96), "==",93)
  &&
  Std.Compare(precentorial.wungee(false,false,16), ">=",14/09/2032)
  &&
  Std.Compare(unnewness.syriac(-15,96), "<",94.1);
}
```

Listing (8.4) Num-Conds-And, NumContractsPerBlock = 4

```
{
  Std.Compare(unnewness.alfonso(2.8,true,10/02/2051), "<",18.25)
  ||
  Std.Compare(unnewness.syriac(-15,9), "==",93.6)
  ||
  Std.Compare(precentorial.wungee(false,false,16), ">=",14/09/2032)
  ||
  Std.Compare(unnewness.syriac(-15,9), "<",94.1);
}
```

Listing (8.5) Num-Conds-Or, NumContractsPerBlock = 4

```
{ exists (2, 4) platycoria.platycoria(platycoria_1),
       Std.Compare(precentorial.zoogonic(78,46,true), "==",true),
       Std.Compare(unnewness.alfonso(2.8,true,10/02/2051), "<",1.25),
       Std.Compare(platycoria.cadmopone(true,01/10/2078), ">",12);
}
```

Listing (8.6) Exists-N-Div-2-N

```
{ exists (1, 4) precentorial.precentorial(precentorial_0),
       cairned.cairned(cairned_1),
       Std.Compare(platycoria.xylophagus(-27,true), ">",85),
       inthronization.inthronization(inthronization_1);
}
```

Listing (8.7) Exists-1-To-N

Figure 8.1: Condition Class Examples

Summary Stats	Min, Arithmetic Mean, Max, Standard Deviation, 99.9% Confidence Interval
Memory Management	Free Memory, Max memory, Total Memory, GC execution times, GC generation info, Misc implementation- specific GC properties
System	Num Context Values, Num Contexts, Num Exists Blocks

Table 8.2: Data Collected for each Benchmark

in 10 forks of the Java Virtual Machine. Thus, we collected a total of 200 data points per benchmark.

Two types of benchmark were executed:

Ontology Loading and Reasoner Initialisation Measures the time required to load the full ontology into the reasoner, build and insert the i_full_context individual and preprocess $\exists$ statements (see Sections 6.5.3 and 6.5.2.6 respectively)

Message Evaluation Measures the time taken to evaluate all loaded Messages

Each benchmark accepts two main parameters:

- A condition class
- The number of Messages to evaluate

For each condition class a benchmark is executed multiple times with an increasing number of Messages specified each time. This allowed us to observe how the number of Messages affects timing, memory usage and other aspects of our system.

A significant amount of additional data is generated during the execution of each benchmark. These are summarised in Table 8.2.

We make no attempts to use regression analysis to fit our data to a family of curves as the number of data points is insufficient for this to be meaningful.

We discuss our findings in the following sections. Please note that in all timing graphs the Y axis units are specified in seconds for initialisation times and milliseconds for evaluation times. These graphs also present the standard deviation for each data point. In most cases these are very small.

8.3.1 Expected Values

This section discusses our findings for the expected data set. We have evaluated our data in terms of initialisation time, Message evaluation time, memory consumption and other factors.

8.3.1.1 Initialisation

Figure 8.2 presents initialisation times with all possible skews. Initialisation times rise as the skew is increased and drop as the skew is decreased. This is the expected result as increasing or decreasing the skew results in the creation of more or less complex conditions respectively.

The expected-0.5 dataset represents the collection of conditions without skew - i.e. these conditions have the properties that we would expect conditions to have during deployment. The initialisation time for this set is reasonably fast when evaluating up to 120 messages, taking 5.62 seconds. However, this time increases at a non-linear rate and soon becomes prohibitive. When initialising 500 messages evaluation takes nearly two minutes and just over nine minutes when initialising 1000 messages. A similar pattern can be seen for all the other "expected" sets, with the curve becoming steeper as the skew increases.

8.3.1.2 Message Evaluation

Our evaluation benchmarks determine the amount of time required to evaluate all of the loaded Messages. We then divide this value by the number of Messages to obtain the average amount of time per Message, with the assumption that each Message takes the same amount of time to evaluate. These results are presented in Figure 8.3. Again, the timings vary appropriately depending on the value of the skew parameter and the curves are non-linear and increase in steepness with the skew value. Evaluation times are reasonable, with expected-0.5 achieving an evaluation time of about 1.4 seconds per Message when 500 Messages are loaded into the reasoner.

8.3.1.3 Memory Requirements

The memory requirements for the expected condition class are quite steep, requiring just under 8 GB of memory to store 2000 Messages with very simple conditions. As expected, memory requirements get progressively steeper as the conditions increase in complexity, and when the skew is 0.9 there is a requirement of nearly 4 GB to represent just 200 Messages.

This steep requirement can be explained in part by the number of ContextValue tuples which need to be represented. However, as we later discuss in Section 8.3.2.3 memory usage remains fairly high even when the number of stored ContextValues is reduced significantly.

8.3.1.4 Context Values and ∃ Statements

We observed that the execution times closely followed the number of ContextValue tuples and ∃ statements loaded into the reasoner. This behaviour is reasonable as the skew value has a direct impact on the number of Contracts

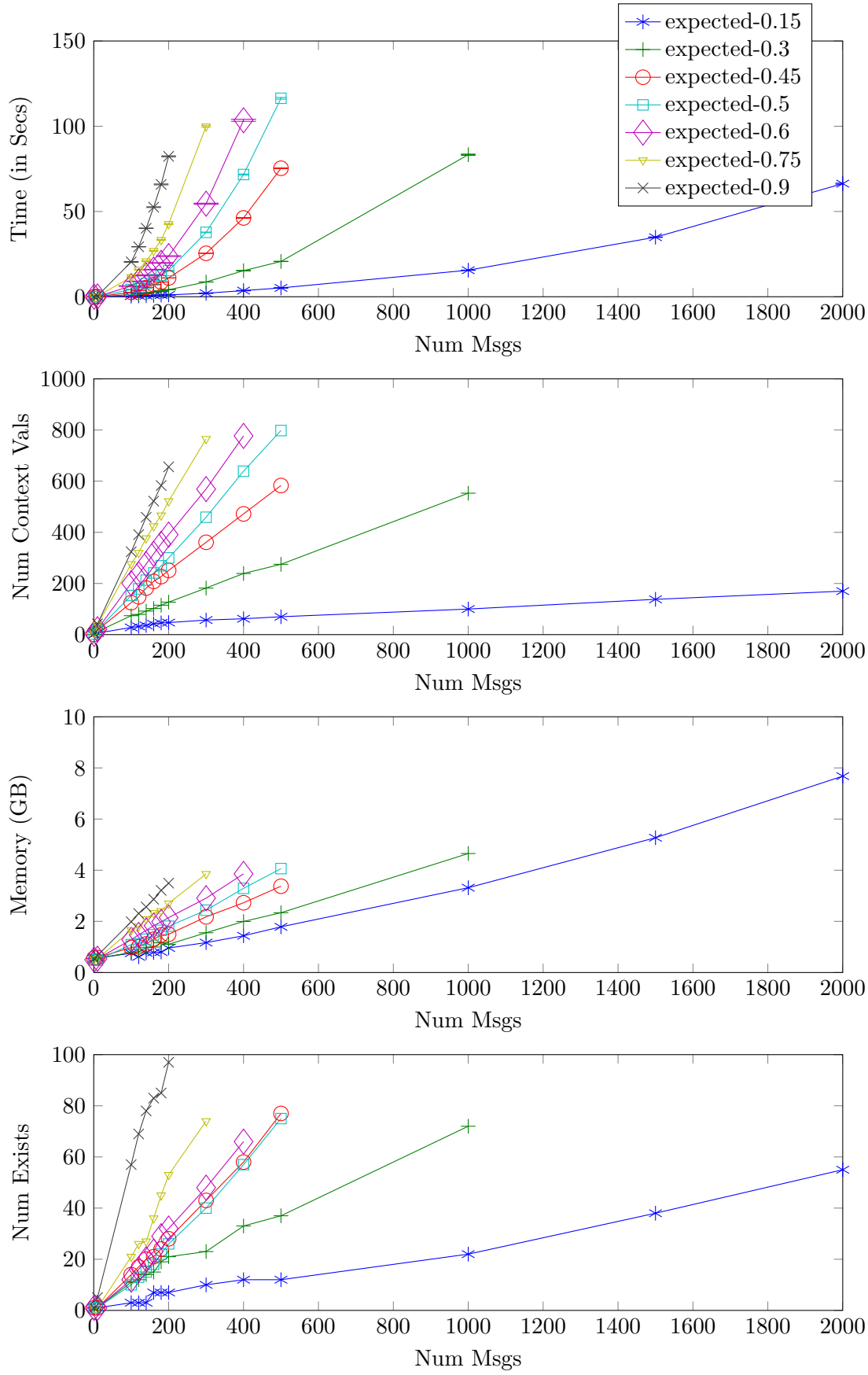


Figure 8.2: Expected Values Initialisation

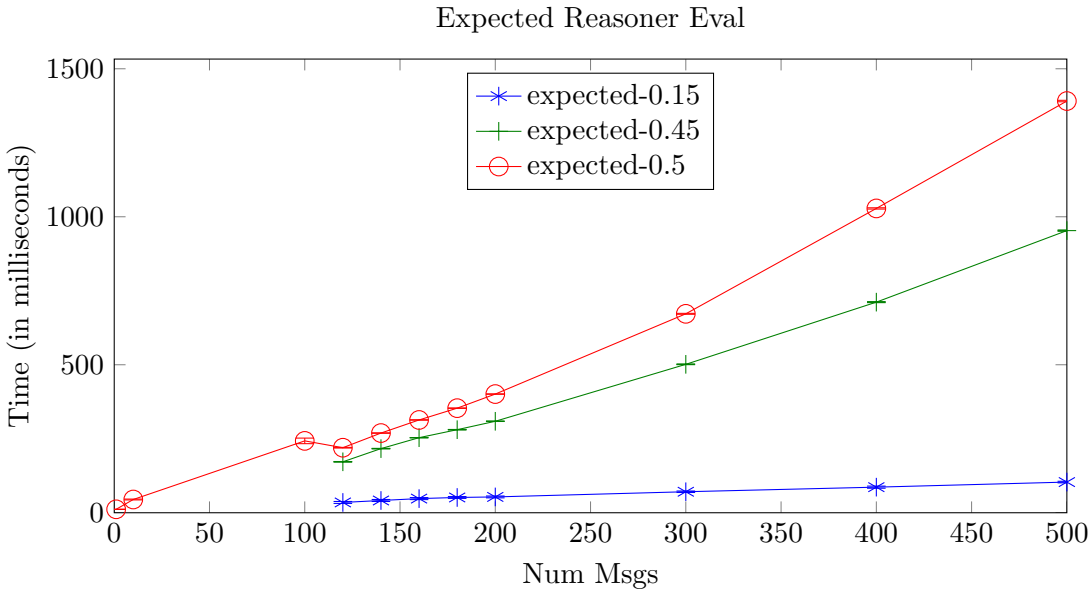


Figure 8.3: Expected Values Evaluation

generated for each condition and the expected number of parameters each Contract within a Context will use. Contract parameter types are generated according to the probability distribution given in Table 7.3 but parameter values for each Contract within a condition are generated using a uniform distribution, meaning that all values are equally likely. Although ontology and boolean parameters have a small number of possible values, Integer, Float, Date and Time types have a very large number, making it unlikely that the same value will be chosen more than once during condition generation. Thus, an increase in the number of Contracts within each condition, and number of parameters for each Contract will, with very high probability result in the generation of a large number of ContextValue tuples in proportion with the number of Messages. The increase in $\exists$ statements is a direct result of the increase in Messages. We calculated the correlation between the execution time and both number of Context Values and number of $\exists$ statements to corroborate our claim. These are available in subtables 8.3a and 8.3b.

In order to eliminate the effect that Contract parameters and $\exists$ statements have on execution times we created the expected-restricted condition class which eliminates parameters and $\exists$ blocks from condition generation. We discuss the effect of these changes in the following section.

8.3.2 Restricted Expected Values

Without parameters the number of ContextValues required to evaluate every Message peaks at 32 for each expected class. This is because each data set

	Context Vals.	∃ Statements
expected-0.15	0.93	0.98
expected-0.3	0.96	0.95
expected-0.45	0.93	0.96
expected-0.5	0.95	0.97
expected-0.6	0.93	0.95
expected-0.75	0.92	0.94
expected-0.9	0.96	0.91

(a) Correlations with Init Execution Time

	Context Vals.	∃ Statements
expected-0.15	0.99	0.94
expected-0.45	0.99	0.99
expected-0.5	0.99	0.99

(b) Correlations with Eval Execution Time

	Context Vals.
restricted-expected-0.15	0.51
restricted-expected-0.3	0.24
restricted-expected-0.45	0.24

(c) Correlations with Init Execution Time

	Context Vals.
restricted-expected-0.15	0.89
restricted-expected-0.3	0.37
restricted-expected-0.45	0.53

(d) Correlations with Eval Execution Time

	0.15	0.3	0.45	0.75
200	33.69	59.57	66.29	72.89
300	30.07	65.01	72.68	78.91
400	40.35	69.53	76.58	
500	37.69	69.14	80.88	
1000	**34.23**	**77.14**	**88.95**	**89.53**

(e) % Increase Initialisation

	0.15	0.45
200	35.29	86.5
300	51.19	91.27
400	57.45	93.66
500	**65.2**	**95.18**

(f) % Increase Evaluation

Table 8.3: Correlation and % Increase Data

uses 8 Contexts, with 4 Contracts each; meaning that without parameters the maximum number of ContextValues needed to represent all of the available Contracts is 32. ∃ blocks are not created so the ∃ preprocessing step is unnecessary. This has a very noticeable effect on both initialisation and evaluation times which we now discuss.

We note that this data set shows a far weaker correlation with number of ContextValue tuples and ∃ statements than the "expected" data set; presented in subtables 8.3c and 8.3d. This suggests that the following results are less affected by the presence of the ContextValues.

8.3.2.1 Initialisation

Figure 8.4 presents the restricted-expected initialisation times. They are a very significant improvement on the times exhibited for the expected class in Figure 8.2. The speed improvement generally increases both as the number of loaded Messages and as the complexity of said Messages increases. This is demonstrated in Subtables 8.3e and 8.3f where the execution times are compared via a percentage increase in speed.

8.3.2.2 Message Evaluation

Figure 8.5 presents the evaluation times. As with initialisation times, there are tremendous speed increases of up to 95.18% when comparing the 0.45 classes with 500 loaded Messages. As the complexity of Messages grow we begin to see progressively steeper curves. However, in general evaluation scales well. For example, we can evaluate conditions with a skew of 0.9 and a load of 500 Messages in approximately one-tenth of a second each.

8.3.2.3 Memory Requirements

The amount of memory consumed is reduced from that consumed by the expected condition class, and the reduction increases as message complexity increases. This seems reasonable as complex conditions tend to have a higher number of Contracts within each condition and thus require a higher number of Context Values. However, despite these reductions the memory consumed remains fairly high. For example, to represent 2000 Messages with a skew of 0.3 we require over 8 GB of memory. This skew produces relatively simple conditions so we might expect a lower memory consumption. When the skew reaches 0.75 we require approximately 6 GB to represent only 1000 Messages.

This could result in scalability issues if the expected number of Messages is high or unevenly distributed within the Regional nodes of the network.

We note that the drops in memory usage at the higher ends of the expected-0.3 and expected-0.45 benchmarks are due to the execution of the Java garbage collector.

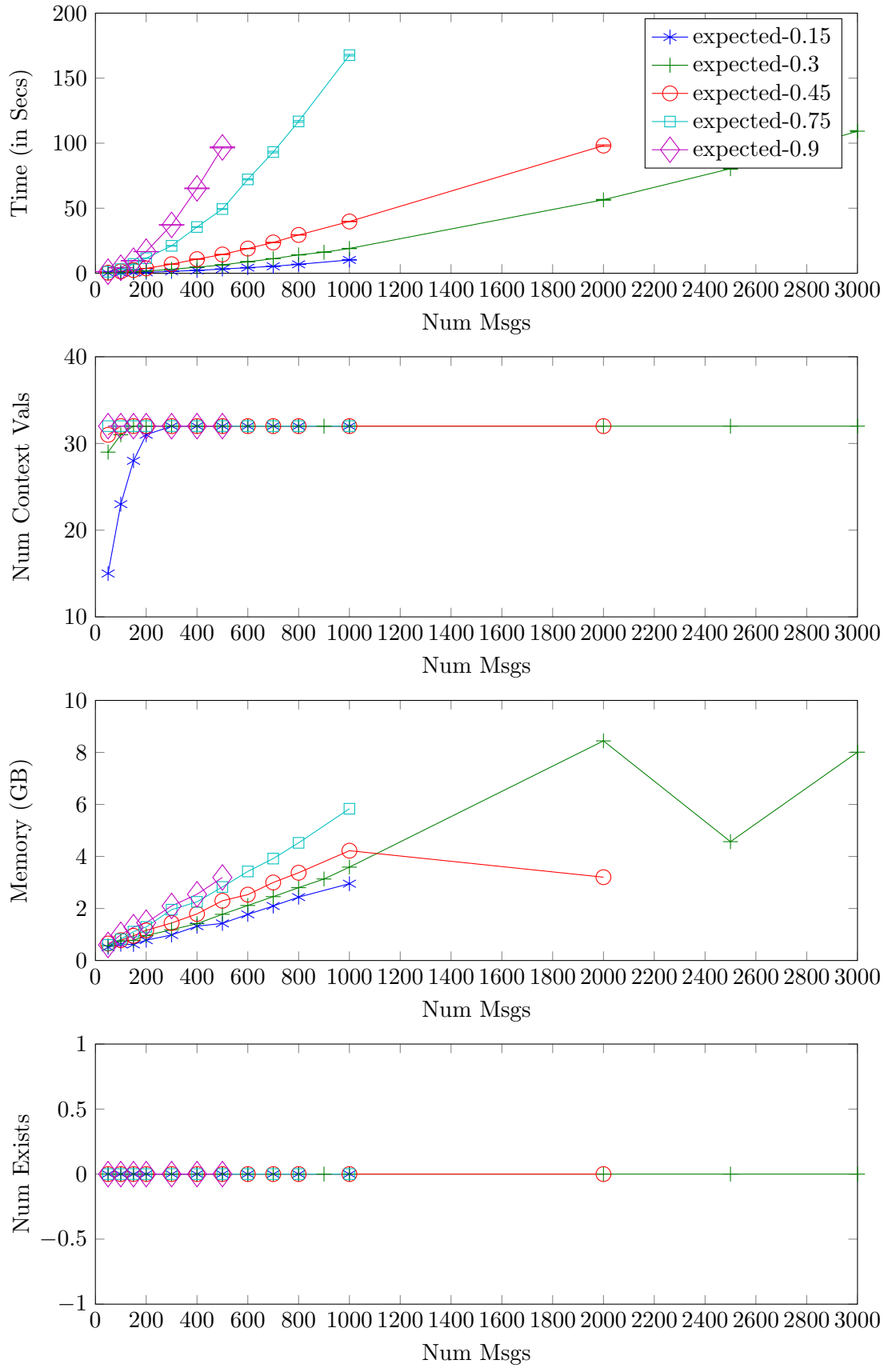


Figure 8.4: Restricted Expected Values Initialisation

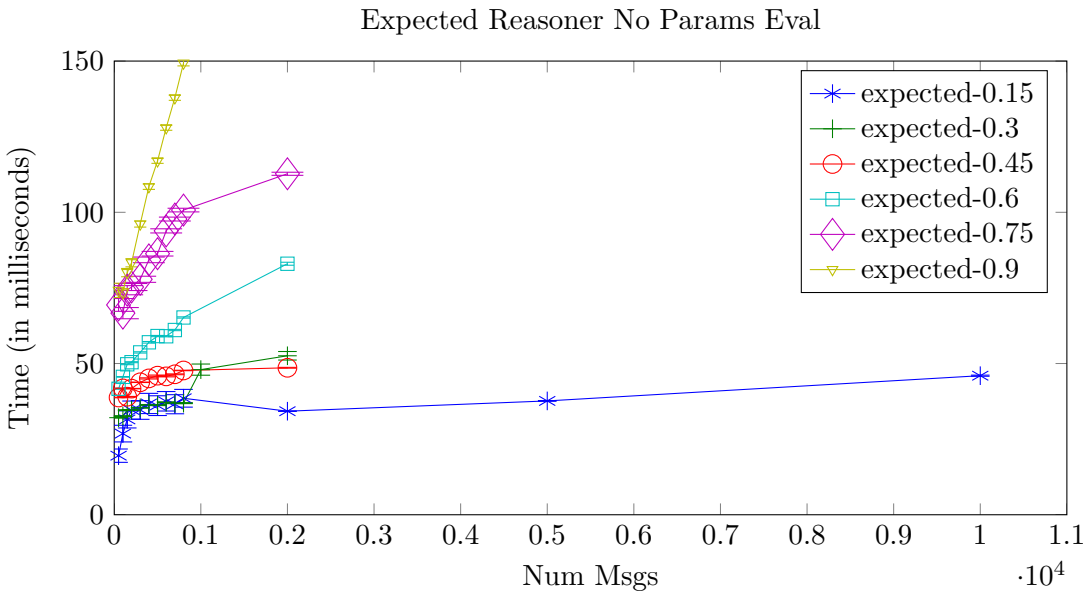


Figure 8.5: Restricted Expected Values Evaluation

8.3.3 Number of Contexts

In addition to taking care with the number of parameters given to each Contract we must also be mindful of the number of Contracts we provide. Aside from the fact that an increase in the number of Contracts will lead to an increase in the dimensionality of the distributed spatial index, it will also result in an increase to the number of possible Contract Values. In addition, whenever a Context or ConcreteContext is defined it is represented in the ontology to allow Regional nodes to lookup a list of bound Participants supporting a given Context (see Section 6.5.1). Thus, the definition of many Contexts and ConcreteContexts will increase the size of the ontology.

Figure 8.6 demonstrates that initialisation times increase when a larger number of Contexts are used. Each data set produces n Contexts and n Contracts within each Context. For example, numContexts-8 will produce a total of 64 Contracts divided evenly between 8 Contexts. However, the effect of introducing many Contracts can be minimal provided that the number and types of parameter for each Contract is carefully considered.

8.3.4 ∃ Blocks

Figure 8.7 illustrates a possibly unforeseen implication of our ∃ preprocessing algorithm in regards to initialisation time. As can be seen, exists1ToN conditions tend to take longer to initialise than their existsN2 counterparts. This is because exists1ToN conditions require fewer Contracts to pass in order for

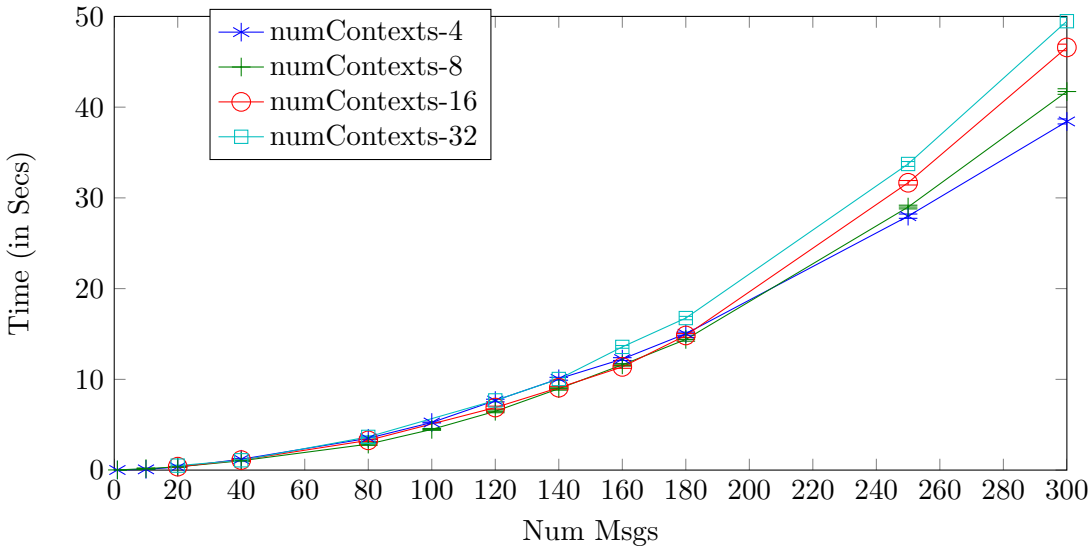


Figure 8.6: Number of Contexts

the ∃ block to succeed; and thus it is more likely that an ExistsResult individual will need to be created and added to the ontology. The ∃ preprocessing algorithm is discussed in detail in Section 6.5.2.6.

This effect can be seen particularly prominently for existsN2-1 which initialises ∃ blocks of the form ∃ (0, 1). In this case, every condition will necessarily pass and require the addition of new individuals to the ontology. These results suggest that this addition to the ontology is significantly more computationally expensive than Contract evaluation as existsN2-1 initialisation times are approximately three times larger that existsN2-4 despite it requiring the evaluation of four times as many Contracts per Message.

8.3.5 Conjunctions and Disjunctions

Figure 8.8 demonstrates that conditions using conjunctions take significantly longer to initialise than conditions using disjunctions. This difference cannot be affected by the number of context values or ∃ statements, or amount of memory use as these are almost identical. The difference gets more pronounced as complexity increases.

8.3.6 Conclusions and Recommendations

Because our language was transformed into OWL all results are intrinsically linked to the performance properties of the OWL language and the utilised reasoner, and thus it is not possible to precisely separate the characteristics of our algorithm from the characteristics of said language and reasoner. Different

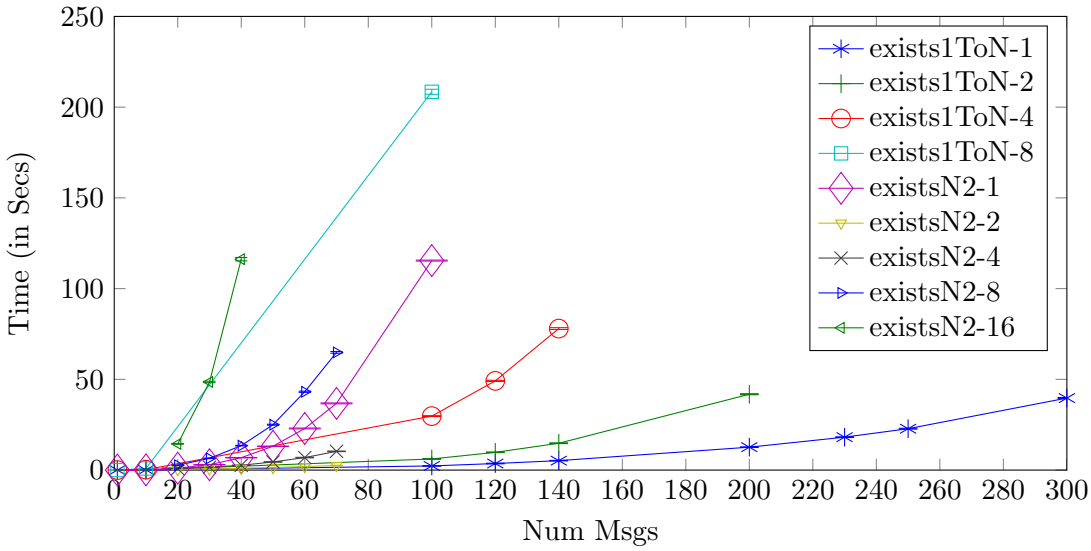


Figure 8.7: ∃ Block Initialisation

reasoner implementations are each likely better suited to a particular class of problem, and these differences would impact our results. For example, in Section 8.3.4 we found that FaCT++ is inefficient when inserting axioms into an existing ontology. This may not be the case when using another reasoner.

However, despite these issues our evaluations did reveal a number of trends and properties of the MediateSpace language:

- The complexity of Contracts has a very significant effect on the performance and memory consumption of our algorithm as the introduction of additional parameters and wide-valued data types result in a greater number of potential ContextValues.
- The OWL representation of our language is quite verbose, resulting in high overall memory consumption.
- Our implementation of the ∃ statement relies on the insertion of an additional axiom to indicate that the statement passes. This resulted in degraded performance for conditions with a small minimum parameter because it increased the likelihood that the ontology would need to be modified. The exact performance impact of this property depends on the reasoner implementation used.

Based on the results and our above discussion we now make a number of recommendations.

8.3.6.1 Context and Contract Definition

Care should be taken over the number of Contexts defined, the number of parameters specified for each Contract and the data type chosen for each

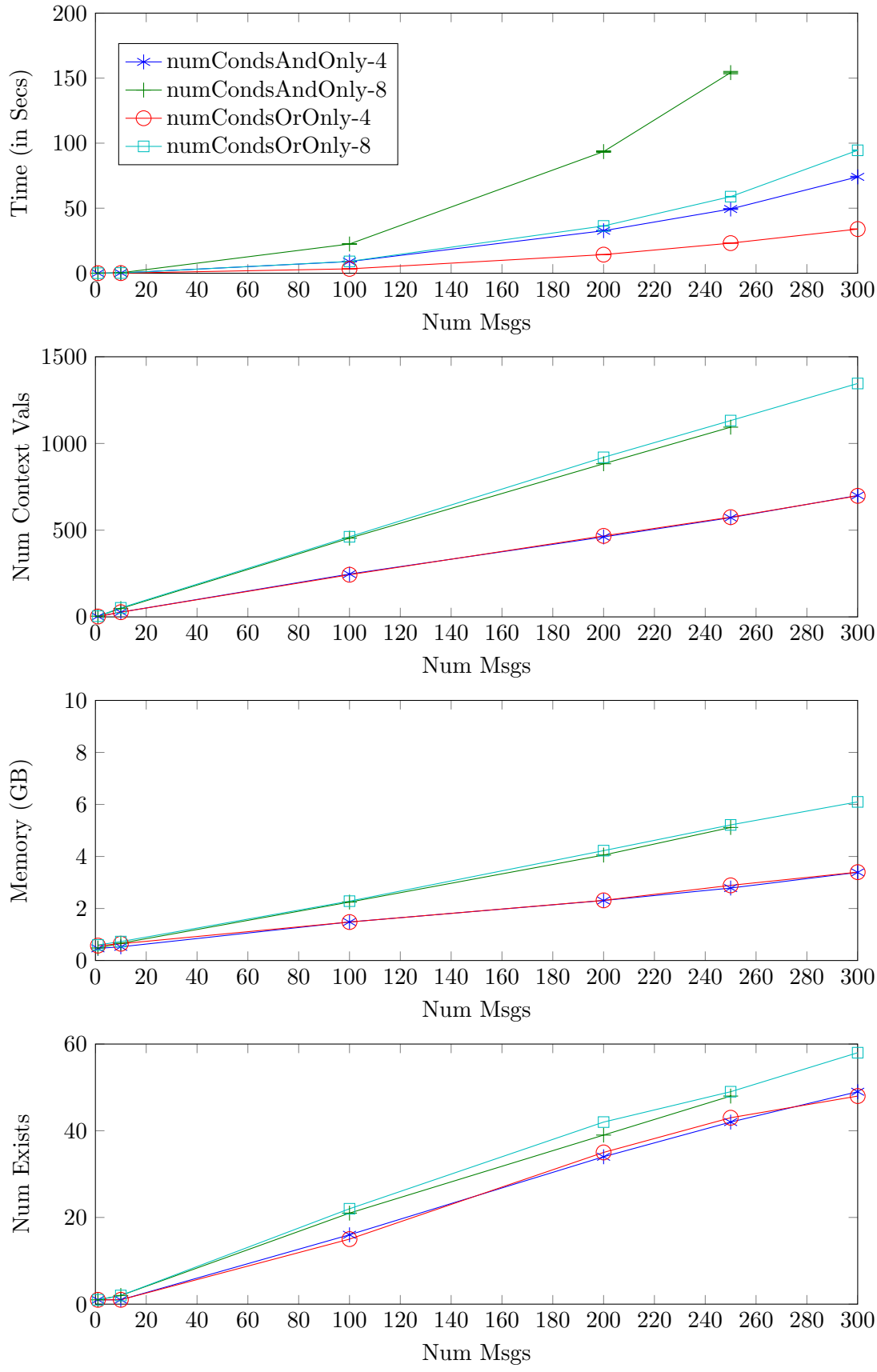


Figure 8.8: Conjunctions and Disjunctions Data

parameter. When Contracts are defined with many parameters this increases the number of different possible combinations of parameter values that can be specified; and thus increases the number of potential ContextValue tuples. Perhaps more important is the choice of parameter data type. If a data type with a wide range of values is chosen, such as Integer or Double, this allows the user to specify thousands of different values.

This could be remedied in a number of ways. These include:

- The introduction of more restricted data types (e.g. a Byte data type that allows the representation of one of only 256 values),
- The implicit limiting of range by documenting a Contract with an allowable range of values,
- The explicit limiting of range by allowing developers to specify constraints on a parameter enforced by the MediateSpace system,
- The removal of wide ranging data types from the language.

Of these options, we believe that the incorporation of more restricted types and a mechanism for restricting value ranges are the most appropriate.

8.3.6.2 Memory Consumption

Memory consumption for even quite simple conditions is fairly high. Thus, Regional nodes should be provisioned with an adequate amount of memory to operate correctly and efficiently. The exact required amount of RAM will depend on the application. Specifically, we should take notice of:

- The kinds of data held in Messages. For example, in our Geocaching application most Messages will likely store a relatively small payload, consisting of co-ordinates and textual information.
- The anticipated complexity of the contextual conditions.
- The Contexts and Contracts supported by the application (discussed in detail in Section 8.3.6.1).
- The density of Regional nodes and the anticipated network load.

Attempts should also be made to reduce the verbosity of our OWL representation.

8.3.6.3 Conjunctions and Disjunctions

Conditions using conjunctions take longer to initialise than conditions using disjunctions. This should be taken into consideration when constructing conditions, but their use cannot realistically be avoided in most cases.

8.3.6.4 $\exists$ Blocks

It is important to consider the effect that $\exists$ blocks have on initialisation time. When the minimum parameter of a block is small it can result in a significant

speed reduction because it becomes more likely that the ontology will need to be modified to signify the success of the ∃ block.

In simple cases it would likely be more efficient to represent a condition using only conjunctions and disjunctions. However, this method quickly becomes infeasible as the conditions grow extremely complex very quickly. This is discussed in more detail in Section 5.3.2.6 and as part of the evaluation of our spatial indexing methodology in Section 8.5.3.

An optimisation could be applied in the simple case where an ∃ block is within the Exists1ToN class by halting the evaluation process when a single Contract evaluates as true. This would work correctly as the block only requires that one Contract passes, and the maximum parameter can be discarded as it is equivalent to specifying a block with no upper bound.

8.4 Simulation

In this section we evaluate our distributed protocol through simulation.

Our simulation runs produce a number of different outputs for post-run analysis, and a significant number of parameters are available which allow us to manipulate the structure of the network and the complexity of the data distributed within it. We will now briefly discuss the simulation output and main parameters before moving on to discuss our findings.

8.4.1 Simulation Outputs

Our simulation produces a number of CSV files containing statistics for every node in the network. This information includes the number of tuples created and received for every type of tuple and response time data for message requests, messages and context requests. We also retain a bind history for all nodes; which consists of a record for each node of the number of nodes it is bound to and the number of nodes that are bound to it. This record is appended to each time the node changes location and hence performs a rebind.

We also generate a graph representing the complete movement and binding history of the network nodes. Binding relationships between two nodes are represented as an edge between them. The graph is represented in the GEFX XML format using the Gephi API. Each node and edge has an associated "spell" of time that stipulates the period that it exists within the network. Whenever a node changes its position its spell of time ends and it is replaced with a new node whose time begins. The same is true for edges whenever a binding changes. In this way we were able to construct complete histories of movement and binding. We used the Gephi [8] network analysis tool to observe the changes in the network over time and to produce several of the graphs presented in this section.

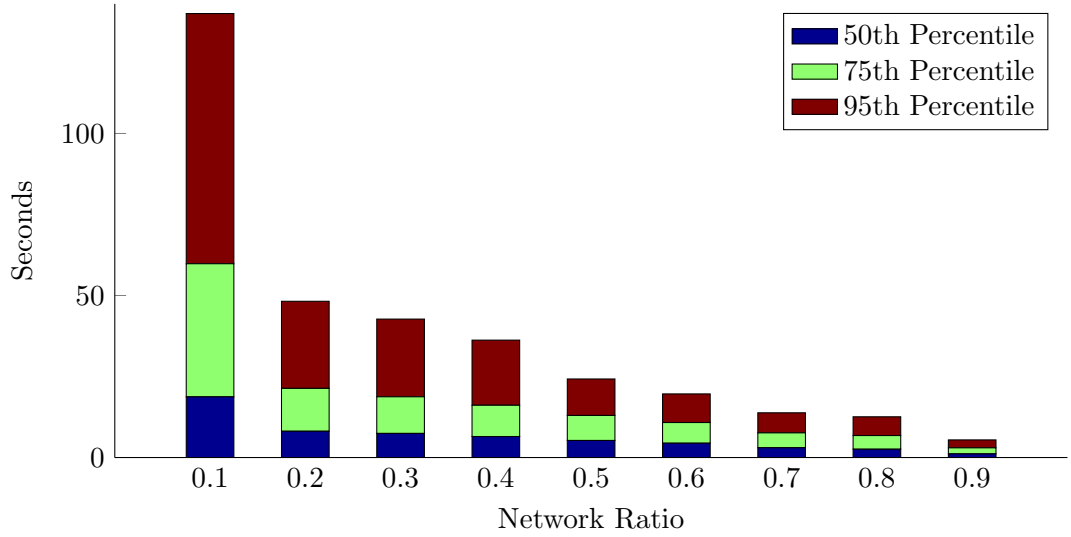


Figure 8.9: First Response Times for Message Requests

8.4.2 Simulation Parameters

We control the ratio of Participant to Regional nodes in each simulation run. The ratio is a floating-point value in the range [0, 1]. When ratio = 0.5, there are an equal number of Regional and Participant nodes in the network. When the ratio > 0.5 the number of Regional nodes outweighs the number of Participants in proportion with the value; and vice versa. For example, if the network contains 100 nodes and the ratio = 0.7, there would be 70 Regional nodes and 30 Participant nodes. The size of the network remains constant regardless of the ratio.

The ratio of Participant to Regional nodes will affect the performance of the network, as when there are a greater number of Participants, Regional nodes will tend to be bound to a larger number of them, and hence will be required to receive and evaluate more network messages.

The remaining parameters are discussed in detail in Section 7.4.

In the following sections we present the details of the simulation runs we carried out and discuss the observations we have derived from the results.

8.4.3 Distribution of Workload

As might be expected, when the node ratio decreases (resulting in a greater number of Participant nodes and fewer Regional nodes) the time it takes to receive a response to a MessageRequest tuple increases. This is illustrated in Figure 8.9.

In addition, we calculated the 50th, 75th and 95th percentiles for the number of MessageMatch tuples created by bound Regional nodes, which we

present in Figure 8.10. We observe that the difference between the 75th and 95th percentiles is very large, suggesting that the distribution of workload is very uneven for a variety of node ratios. The reason for this could be because of the following:

1. Each Regional node was set to allow up to 50 Participants to bind to it.
2. Our node mobility algorithm defines different cities of dramatically decreasing size, and each node resides within one of these cities at startup. The probability of a node residing in a given city is proportional to the size of the city. Thus, the larger the city, the more nodes we would expect to be within it.
3. Participant nodes tend to only travel short distances and also tend to spend much of their time within just a handful of locations.

We suspected that the workload was so unevenly distributed because Participants tended to reside in the big cities and rarely leave them. Thus, a large number of Participants would be bound to a small number of Regional nodes throughout the simulation. This hypothesis can be validated by looking at the graphical history of node movements and bindings. The graph in Figure 8.11 illustrates the relative degree of each Regional node through size. The larger the node, the greater its degree. The edges shown represent the binding relationships between Regional nodes; the nodes and edges for Participant nodes have been omitted for clarity. The largest four of these nodes have a degree in the range [737, 1098]. In the discussed simulations each Regional node performs binding once to a maximum of six nodes at startup and remains static throughout. Thus, we can conclude that node degree is a good measure of the binding behaviour of Participants.

8.4.4 Network Size and Density

The network is very sensitive to the number of nodes present, and more importantly to the geographical distribution of these nodes. This is because shared context has a finite distance where it remains valid so Participants must either have direct access to all the context information they require or they must be situated quite close to other Participants to request context from them. Without the ability to obtain context information, Message candidates cannot be evaluated. Figure 8.12 presents the 50th, 75th and 95th percentiles for the number of Messages received by each Participant in an 100 node network with a node ratio of 0.9. The only variable changed between runs was the maximum geographical area that the nodes could reside in. We can see that the number of received Messages increases as the geographical area gets smaller. The reason for this is that nodes are necessarily closer to one another, and thus can share context information. We note that the 95th percentiles are much larger than the 75th, once again a consequence of clustering within

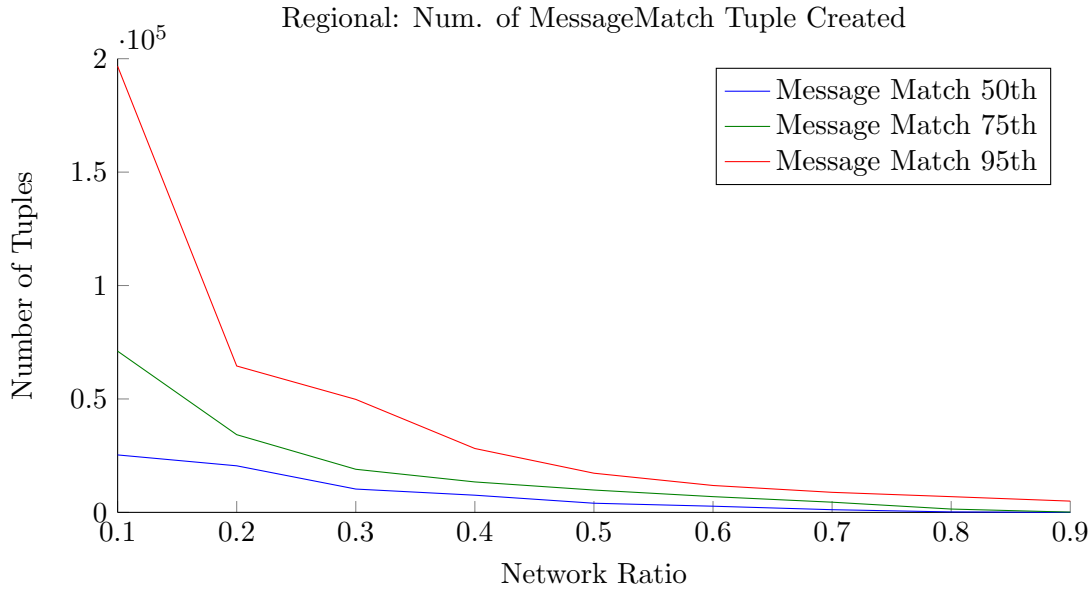


Figure 8.10: Number of MessageMatch tuples dispatched from the bound Regional Node

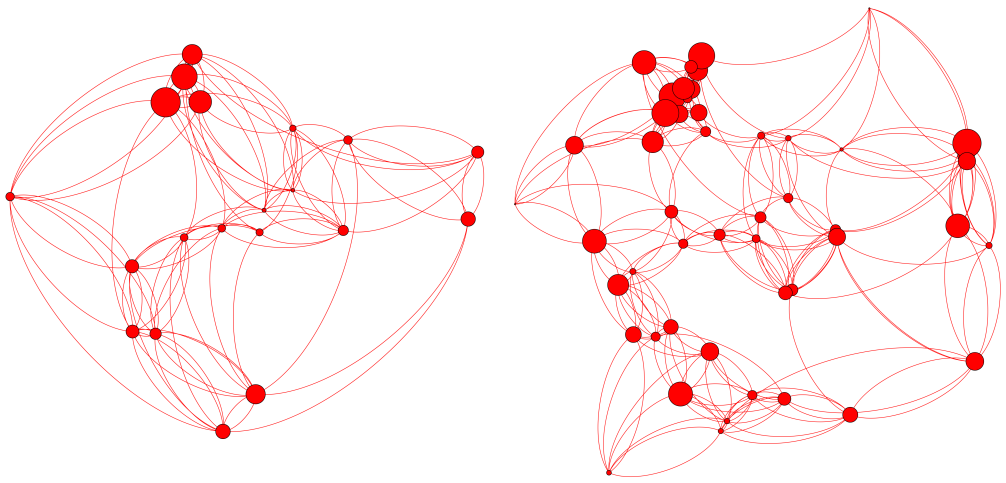


Figure 8.11: Relative Node Degrees when ratio = 0.2 and 0.5 Respectively

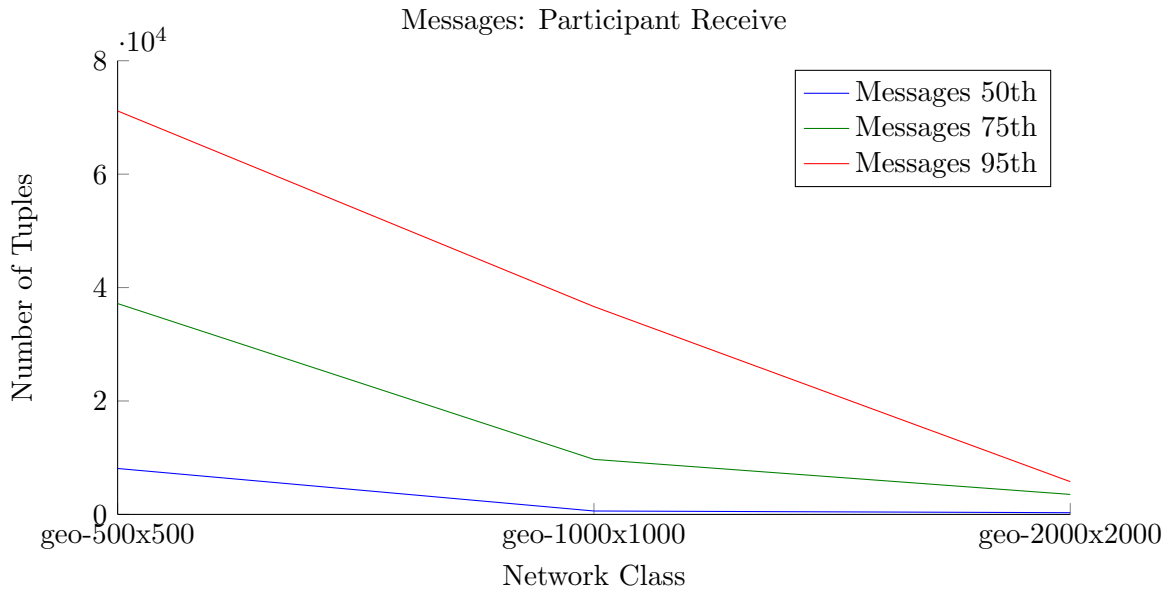


Figure 8.12: Num. of Message Received by Geographical Bounds

large cities.

The numbers and density of Regional nodes are also an important consideration as Messages are stored on the Regional node whose geographical position is closest to the co-ordinates specified within the Message's contextual condition. If Regional nodes are sparsely distributed, a single node may become responsible for most of the Messages within an area. This provides the motivation for the load balancing scheme proposed in Section 4.3.5.5. In our simulations without the load balancing scheme in place we found that in almost all cases Message Requests were only processed by the bound Regional node, and were not forwarded to any other node because all of the applicable Messages were stored on the bound node.

8.4.5 Binding and Unbinding

As the ratio of nodes increase Participant nodes tend to bind and unbind from Regional nodes more often. This is a result of there being a larger number of Regional nodes in the network, making it more likely that Participants will move to an area with a closer Regional node to bind to.

The number of binds made by Regional nodes at the 50th percentile seems to spike when ratio = 0.2 but then settles into a predictable pattern. The spike is a result of an increase in the number of rejections. The spike disappears when the network contains enough Regional nodes to bind to without major contention.

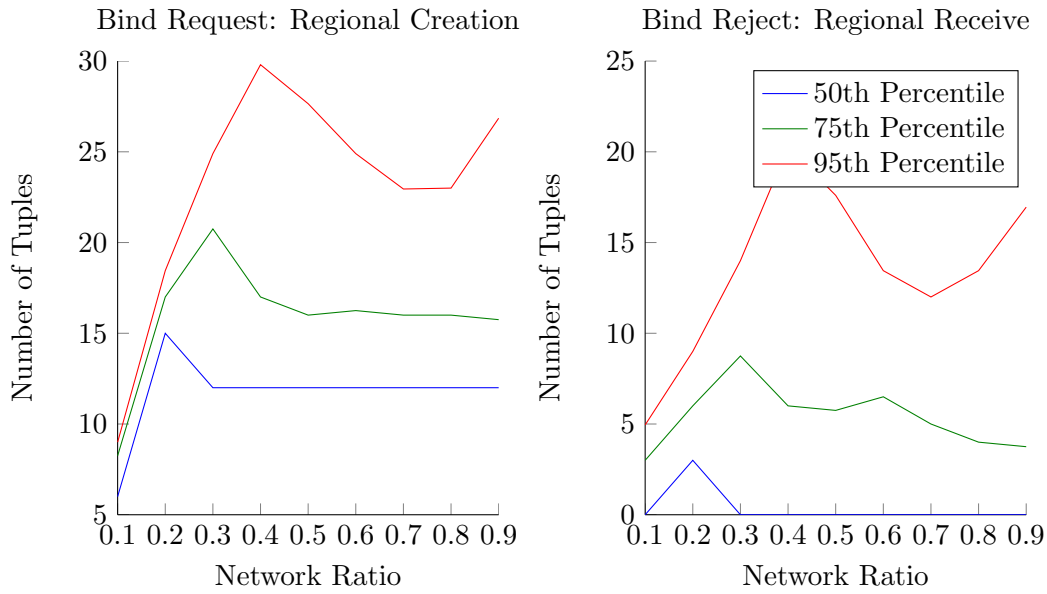


Figure 8.13: Regional Node Bind Request and Reject Behaviour for 50th, 75th and 95th Percentiles

However, an interesting behaviour occurs for the bind behaviour of the 75th and especially 95th percentiles. In this case, the binding behaviour fluctuates between an increase and decrease in bind messages as the node ratio increases. This is a result of a fluctuating number of bind rejections from other Regional nodes, which itself is a consequence of the clustering of nodes within large cities. These nodes will attempt to bind to one another and when a sufficiently large number of Regional nodes are present within the cluster many of the nodes will reach their bind limit before all of their neighbours have completed the bind process - resulting in an increase in rejections. As the number of nodes increases the network reaches a point where the clustered nodes are able to bind with one another again without resulting in a large number of rejections. This pattern then begins again as the number of nodes grows. This is shown in Figure 8.13.

8.4.6 Condition Complexity and Context Requests

From our benchmarking data in the previous section we can see that when parameter values are permitted the number of required Context Values increases sharply as the complexity of conditions increases. If a significant proportion of issued Message Requests have wide ranging queries that consider a large number of candidate Messages, the bound Regional nodes could be responsible for issuing a large number of Context Requests to their neighbours. This is discussed in more detail in Section 8.4.7.

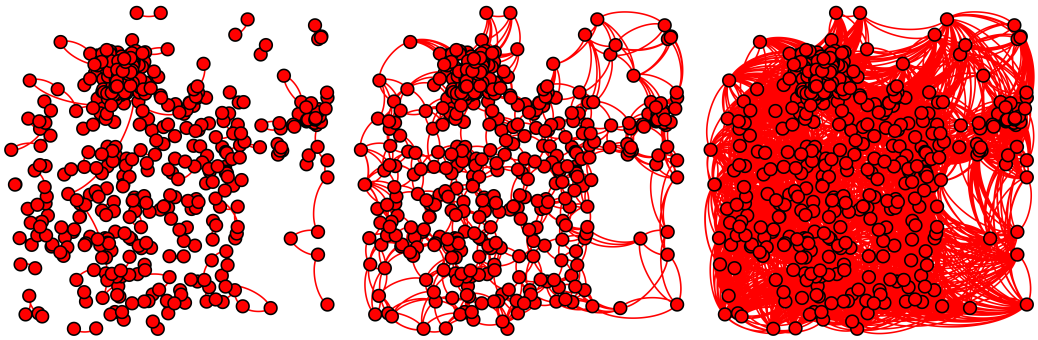


Figure 8.14: Regional Nodes with Maximum Neighbours of {1, 6, 40} respectively

Care must be taken when specifying a maximum valid distance for Contexts. If this is set too low and the network is sparse Participants will rarely be able to request the Context from one of their peers. If, however the distance is set too high and applicable ConcreteContexts are not plentiful, it can result in a context request being dispatched to a large number of Regional nodes before the context is found or the maximum distance is reached. If a large number of requests are made for this Context it could result in part of the network being flooded with requests. In addition to applying the Contract restriction methods outlined in Section 8.3.6.1, the excessive proliferation of Context Requests could be curtailed in a number of ways.

The simplest method would be to allow Regional nodes to only forward a Context Request to their closest bound neighbour. This methodology may work well in cases where there is a strong connectedness between the Regional nodes as each node will only be required to issue a single request for the Context Value, yet the request will be recursively issued amongst each of the nearby nodes and all participating nodes will benefit when the value is obtained. However, when connectedness is poor, requests may go quickly beyond the allowed distance without contacting nearby nodes. The connectedness of nearby Regional nodes is largely dependent on the bind limits specified at each node; when the limits are large there will be a strong connection in an area and vice versa. This issue is illustrated in Figure 8.14. We note that when the maximum number of neighbours was 40, the queues for a number of nodes become saturated and the simulator stopped execution. This demonstrates that there are practical limits to the maximum number we can specify. However, The network is well connected when a maximum of 6 neighbours is specified so this should not cause an issue.

An alternative approach could be to use a stigmergic routing protocol such as that proposed by Fleming et al. [35]. Their work is based on the behaviour of ants, which produce pheromones to encourage other ants to follow their path. The pheromone trail gets stronger as the number of ants following a

given path increases. This was applied to a distributed question-and-answer network where links to desirable network members are strengthened. In their case strengths were applied for each of n categories (e.g. computers and Internet or Sport). In the context of our work Regional nodes could store a link strength for each possible Contract. This approach seems promising for scenarios where Regional nodes are static and long-living as Participants tend to stay within a small, well defined radius of their home (as discussed in Section 7.4.3.2). Thus, it seems reasonable to assume that the calculated link strengths would remain valid over an extended period.

8.4.7 Message Specificity

When users issue Message Requests into the network they can be at varying levels of specificity. Specificity refers to the number of Contracts within the request and the range of values being matched for each Contract. When a Contract is unspecified the system assumes its range to be the largest possible range for its data type. Thus, by specifying a Contract it helps to increase the specificity of the query. We specify two example conditions below assuming the system uses a single Context tuple containing two Contracts. The first condition is very specific as it includes both Contracts and specifies small ranges for each; whereas the second is quite unspecific as it specifies only one of the conditions and has a reasonably large range.

```
{ Std.Compare(B.B1(), ">", 5) && Std.Compare(B.B1(), "<=", 45) &&
  Std.Compare(B.B2(), ">", 28) && Std.Compare(B.B2(), "<", 32) }

{ Std.Compare(B.B1(), ">", 5) && Std.Compare(B.B1(), "<=", 948); }
```

We consider two classes of specificity: Specific and Unspecific. The Specific class has a RangeOfSpecificity parameter value of 1.0 and ForceMatchingContracts = true to ensure that the range of each Contract is restricted from both sides. The Unspecific class has a RangeOfSpecificity value of 0.0 and ForceMatchingContracts = false.

The number of Messages considered and Context Requests issued will increase as conditions becomes less specific; which will increase the response time for a Participant's request. Response times for a network with ratio = 0.9 are given in Figure 8.15. This demonstrates that response times are approximately three times slower when a request is specific, compared to one that is unspecific. Figure 8.16 shows the difference in the number of Context Requests made by Regional nodes and Messages received by Participants respectively. We note that Figure 8.16 shows only the 95th percentile as it was only the top 5% of Regional nodes that dispatched any Context Requests at all. This is because these Regional nodes are the only ones within a valid geographical distance of Participant nodes. This is once again an effect of the big city clustering problem discussed previously.

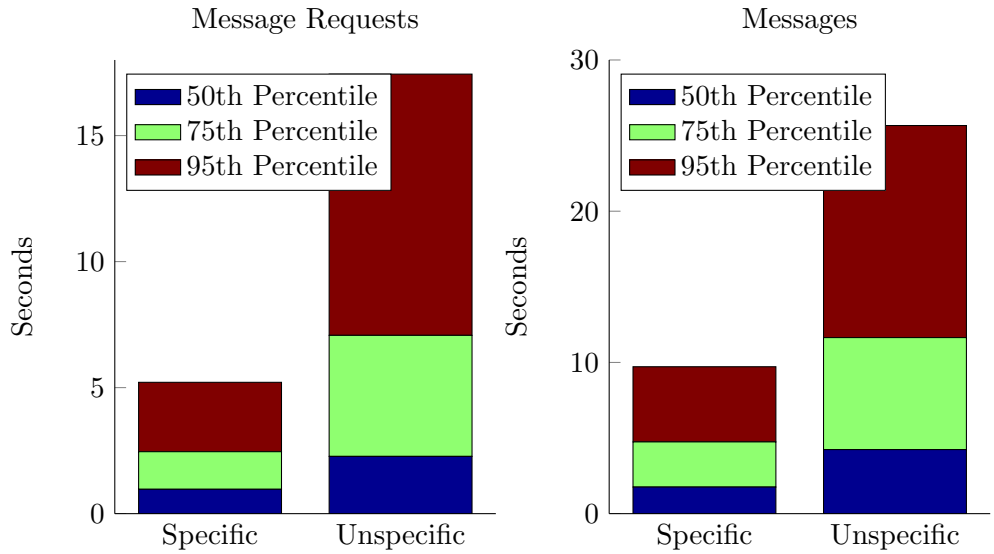


Figure 8.15: First Response Times for Specific and Unspecific Requests

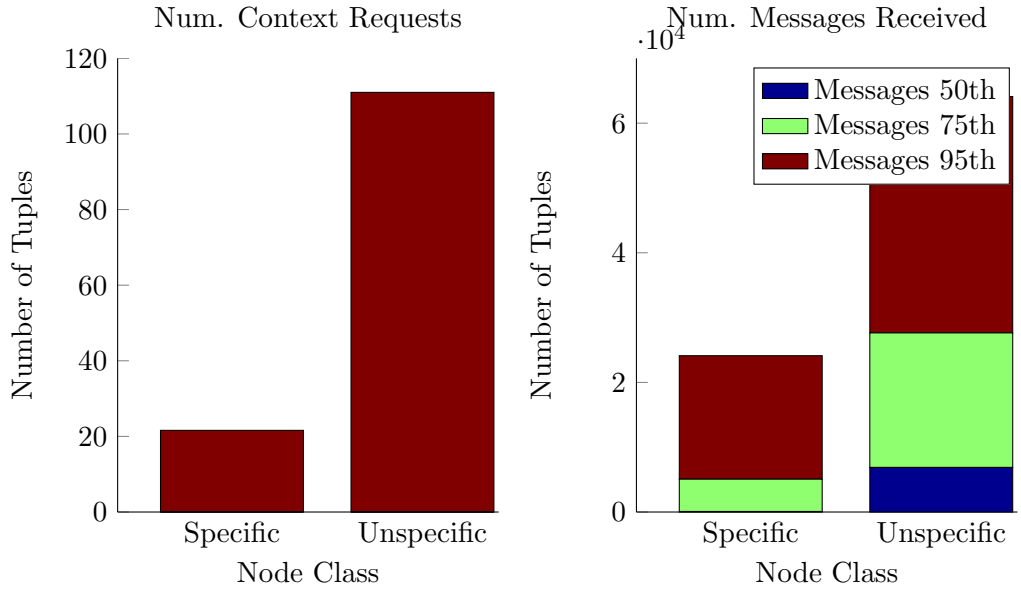


Figure 8.16: Num. Context Requests and Messages Received

8.4.8 Validity of Simulation Results

Care must be taken when making conclusions about response times in a simulated environment as there are a wide variety of factors that can skew the data. Firstly, there may be other processes sharing the processor and memory. Although our simulations were run on an exclusively owned core within the cluster environment discussed in Section 7.5.3, they may have shared memory with the processes of other users.

The other important influence on timing is the simulation environment itself. For an N node network, the PlanetSim simulator is required to instantiate some combination of N Regional and Participant nodes. Although Participant nodes are fairly lightweight, Regional nodes require additional memory and processing time for parsing and evaluating OWL conditions. Thus, in a large network the memory and processor requirements will be much greater than if each node were executed on their own device.

Our response and execution times should be treated only as an approximation. To obtain accurate results we would need to deploy our application on actual devices or on a network environment such as PlanetLab [1]. A deployment on PlanetLab would have its own issues however as it does not provide support for mobile nodes; meaning that this aspect of the tests would still need to be simulated.

8.4.9 Summary

We now provide a number of recommendations based on our above analysis.

- Heavily populated areas require well equipped Regional nodes to handle the demand and ideally multiple Regional nodes should be deployed in reasonable proximity with one another to balance Message storage and lookup and the issuing of Context Requests.
- The density of the network is an important consideration as, if the network is too sparsely distributed Participants will be unable to share Context information with one another. Thus, in order to be effective, the network would require quite high adoption rates within any area it is used.
- In order to be able to properly provision the Regional nodes in the network, we must have an understanding of the typical complexity and specificity of the contextual queries specified. The more complex and unspecific queries become, the better the specification and number of Regional nodes required.

[1] https://www.planet-lab.org/

8.5 Properties of Spatial Indexes

We now discuss the generation of spatial indexes and the factors that dictate the number of indexes created for a given condition. Firstly we describe the data sets used in our analysis and then follow this with a discussion of our analysis of the expected, num-conds-and, num-conds-or, exists-n-div-2-n and exists-1-to-n condition classes.

8.5.1 Data Sets

We performed analysis on five data sets containing 1000 conditions each for each condition class. The sets were generated to establish the number of indexes we would expect to generate for each condition. Each set modifies the probability of choosing a conjunction over a disjunction each time a connective is requested during the condition generation process. They are as follows:

and-1-0-max-100 100% probability of choosing a conjunction over a disjunction with the maximum ∃ threshold set to 100,

and-0-8 80% probability of choosing a conjunction over a disjunction,

and-0-5 50% probability of choosing a conjunction over a disjunction,

and-0-0 100% probability of choosing a disjunction over a conjuction,

and-1-0 100% probability of choosing a conjunction over a disjunction.

The last four sets all have a maximum ∃ threshold of 10,000. Thresholds are discussed in Section 8.5.3 and in more detail in Section 5.3.2.6.

8.5.2 Expected Values

Figure 8.17 presents our analysis on the "expected" class of conditions. We have taken the 50th, 75th and 95th percentile. As expected, the number of indexes tends to increase as condition complexity increases. The number of indexes for expected-0.5 (the class we anticipate best represents reality) peaks at five indexes when disjunctions are used exclusively and drops to one when only conjunctions are used. The maximum number of generated indexes at the 95th percentile for the "expected" class with all skews is six which we believe to be reasonable.

8.5.3 ∃ Blocks

Our algorithm tends to produce far more spatial indexes for ∃ blocks than for other connectives. This is because, depending on the number of Contracts and the minimum and maximum parameters there can be a very large number of

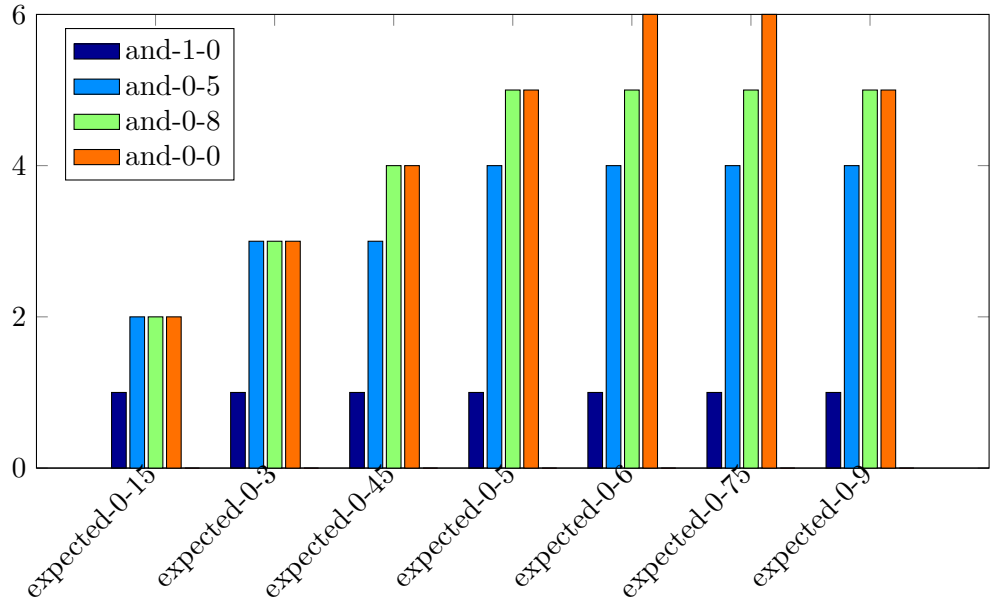


Figure 8.17: Number of Indexes at 95th percentile

combinations that satisfy the condition. For this reason we enforce a maximum number of indexes per ∃ block. This is discussed in more detail in Section 5.3.2.6.

This throttling algorithm can counterintuitively result in more complicated ∃ blocks having fewer generated indexes than a simpler ∃ block. This is because the algorithm continually simplifies the ∃ block parameters until the number of generated indexes falls below a threshold. Thus, a more complicated condition may need to be restricted much further than a less complicated condition, resulting in the latter generating a larger number of indexes.

Figure 8.18 presents the number of generated indexes for each of our ∃ condition classes with the maximum index thresholds set to 100 and 10,000. The use of two wildly different thresholds allow us to demonstrate the effects of our throttling algorithm.

When the number of Contracts within each condition is small the number of indexes is identical for both thresholds as throttling has not come into effect. However, as the number of Contracts grows the difference between the thresholds becomes very significant. Note that the values for exists-1-To-N-16 are identical as even with a threshold of 10,000 it is not possible to represent all indexes.

This exponential increase in the number of indexes can be avoided for the exists-1-To-N class by using the optimisation discussed in Section 5.3.2.6 but this remains an issue for other classes of ∃.

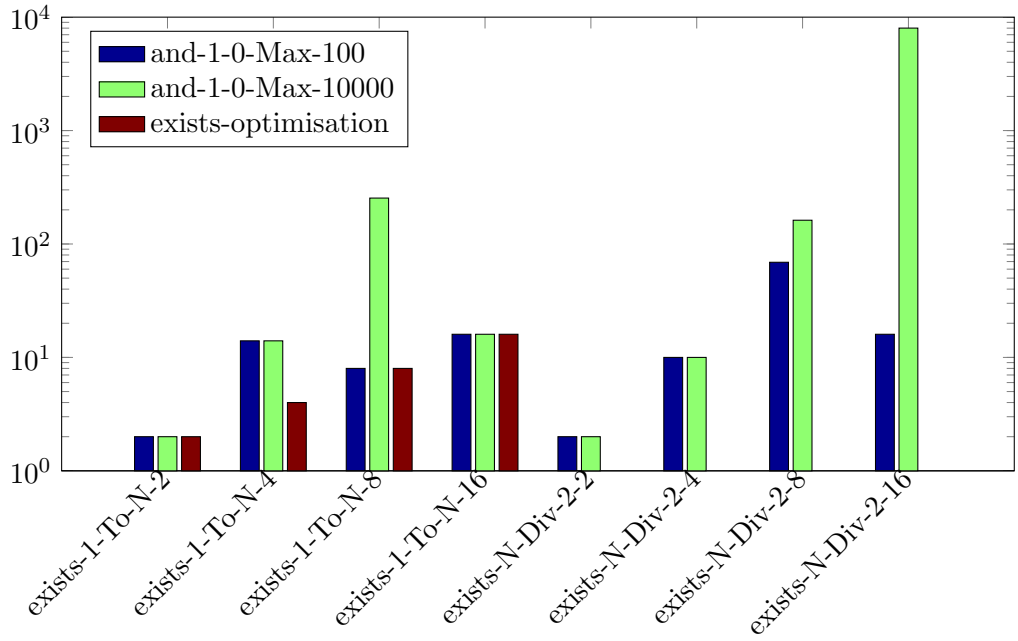


Figure 8.18: Exists Indexes

8.5.4 Combining Conjunctions, Disjunctions and ∃ Blocks

Figure 8.19 presents the index generation statistics for the num-conds-and and num-conds-or classes with a ∃ threshold of 10,000. At the 75th percentile the maximum number of indexes is only 22, generated when only disjunctions are used and each condition contains 16 Contracts. Very large values appear at the 95th percentile which account for the complex ∃ blocks within some of the conditions. This is encouraging because it suggests that complex ∃ tend to be only a small percentage of the data and that their incorporation may not be as problematic as Section 8.5.3 suggests.

Although the outliers are of concern the majority of the data is within what we deem a reasonable range. If ∃ blocks became an issue one remedy could be to reduce the index threshold, which would result in more false positives but crucially no false negatives.

8.6 Summary

The preceding chapter discussed the results of our evaluation.

We divided our benchmark tests into initialisation and message evaluation times. While we found evaluation times to be reasonable and scalable, initialisation times for the “expected” class tend to be slower than desirable. The main cause of these inefficiencies are the large number of required Con-

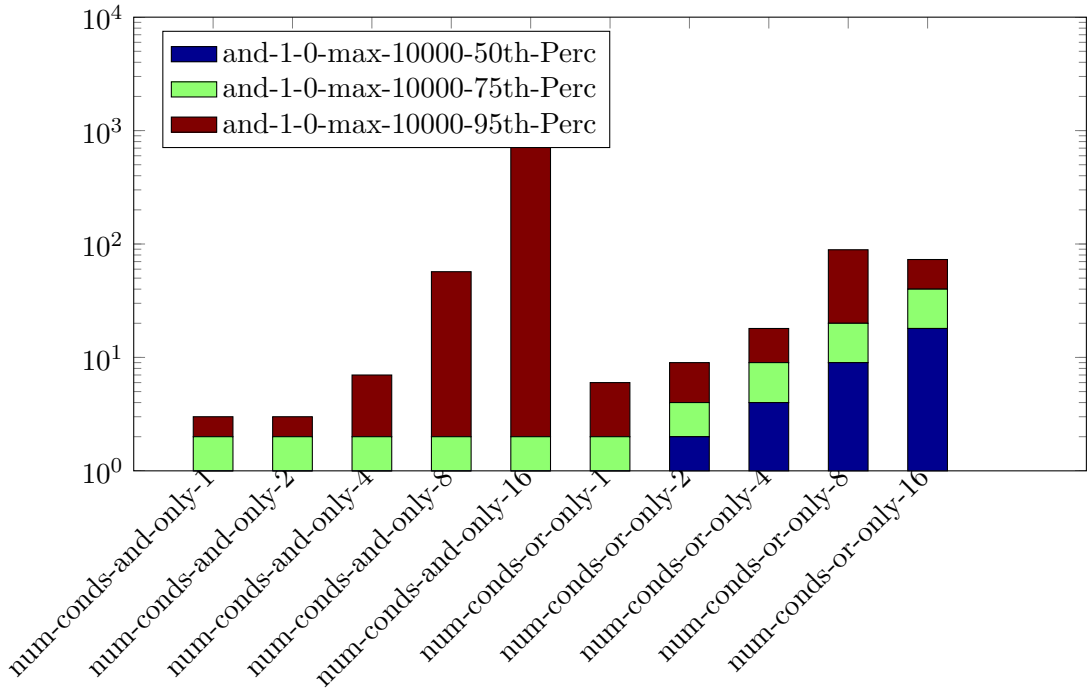


Figure 8.19: Number of Indexes at the 50th, 75th and 95th Percentiles

textValues and the preprocessing required for exists blocks. This hypothesis is supported by our "restricted expected" results which show very significant speed increases due to a reduction in the number of ContextValues and the elimination of exists blocks. We propose that the number of ContextValues can be reduced by restricting the number and types of parameters given to Contracts or by implicitly or explicitly limiting the ranges of values each parameter may take.

We also found the memory requirements for even quite simple conditions to be of concern, requiring Regional nodes to be provisioned with significant amounts of RAM. Attempts should also be made to alleviate this requirement by reducing the verbosity of our OWL representation.

In our simulations we found that the distribution of workload was very uneven because nodes tend to cluster within large cities; meaning that the majority of communication is localised within these areas. We also found that node density had a dramatic impact on the number of received messages as nodes within sparse areas were unable to obtain valid context information. Finally, we demonstrated the impact that message specificity has on response times, the number of generated Context Requests and the number of Messages received by Participants.

Our evaluation of the spatial index generation algorithm found that the number of indexes generated for each condition tends to be low, but that

certain configurations of ∃ blocks can result in the generation of a very large number of indexes. We have proposed a throttling algorithm to limit this at the cost of some expressivity.

The main take-home message is threefold:

1. The MediateSpace middleware is best suited to urban areas where there will be a higher density of Regional and Participant nodes. This density of nodes allows Participants to share ContextValues freely.
2. The required density of and provisioning needs of Regional nodes is strongly related to the complexity and specificity of contextual conditions. As condition complexity increases so does the required Regional node density and/or provisioning needs, whereas in the case of specificity the opposite is true, with the required density and provisioning needs decreasing as specificity increases.
3. Insufficient provisioning can be compensated for by increasing node density in the affected areas, and insufficient node density can be alleviated by improving the specification of existing nodes; provided that the density is sufficient to allow Participants to share ContextValues.

9 Conclusions and Future Work

9.1 Introduction

The goal of this thesis was to design, implement and evaluate a decentralised context-aware middleware for the distribution of information. Information is packaged within Message tuples which stipulate a contextual condition that must be met by any user wishing to receive it.

To this end, we developed a context-aware language for communication between devices and the specification of complex contextual conditions. We also constructed a network protocol, an indexing algorithm to allow users to efficiently lookup relevant Messages and an OWL representation of our condition language to allow the evaluation of Messages.

The main contributions of this work are as follows:

- **A Context-Aware Language**
  For supporting the development of Context-Aware applications.

- **A Context-Aware Middleware**
  A distributed and scalable context-aware content distribution middleware.

- **Contextual Condition Spatial Indexing Algorithm**
  An algorithm for mapping our contextual condition language to a multi-dimensional spatial index such as the R-Tree [46].

- **Contextual Language OWL Representation**
  An OWL representation of our context-aware language

- **A Context-Aware Framework Taxonomy**
  A taxonomy for comparing frameworks along the dimensions of flexible evaluation, ontology extension, heterogeneous interoperability and

decentralisation.

We now provide a short summary of all preceding chapters and then move on to a discussion of future work.

9.2 Thesis Summary

In Chapter 2 we provided a detailed examination of the pertinent decentralised protocols and justified our choice of a tuple-space based approach with a distributed spatial index for Message lookup.

Chapter 3 presented an overview of context modelling techniques and a discussion of current context-aware middleware solutions. We chose to use an OWL based representation for context evaluation because of the inference mechanisms it provides and set out four criteria for the design of our middleware. These were *Flexible Evaluation*, *Ontology Extension*, *Heterogenous Interoperability* and *Decentralisation*.

We provided flexible evaluation through our contextual condition language which provides support for a modified form of universal and existential quantification ($\forall$, $\exists$), conjunctions, disjunctions and negation. Block scopes and nested blocks are also supported so conditions can be arbitrarily complex.

Ontology extensibility was supported through our Context and ConcreteContext tuples. Context tuples allow a common interface to be provided for a type of context and ConcreteContexts allow the specific implementation for a given sensor. Ontologies may be specified within a Context if required, with the ConcreteContext tasked to provide the correct mappings from raw sensor data to the ontology concepts.

We provided heterogenous extensibility through the definition of our context-aware language. These structures can be shared with other devices in the network over any communication protocol such as SOAP and require only that the device is able to parse the language.

Our network protocol is decentralised in nature; splitting computational tasks and message storage and lookup across the nodes of the network. Our decentralised network was designed to split nodes into two categories: Participant and Regional.

Regional nodes are responsible for all computationally expensive tasks (such as the evaluation of contextual conditions) and for communication with other remote nodes to obtain context information or forward requests for information.

Participant nodes can issue Message Request tuples to lookup and obtain relevant messages, with their only other main responsibility being to provide context information when requested. The network is location-aware, with nodes binding to their most geographically proximate neighbours, which allows Regional nodes to ensure that any context information obtained from a remote node is not so far away from the requester as to have become invalid.

Chapter 4 motivated our middleware by way of a pervasive advertising application and provided a detailed discussion of our context-aware language and network protocols.

Chapter 5 described the mapping between our language and spatial indexes and Chapter 6 described our methodology for translating our language into an OWL representation for evaluation.

Chapter 7 discussed the high level design for our middleware and explains the methodology we followed during the evaluation of our system. This includes the parameters used for context modelling, simulation and benchmarking.

This leads into our results in Chapter 8. Firstly, we evaluated our OWL representation in terms of execution speed and memory consumption. This is followed by an evaluation of our network protocol in terms of response times and other concerns such as the distribution of workload and data across the network. Finally, we evaluated our spatial indexing algorithm in terms of the number of indexes required to represent conditions of varying complexity.

Our evaluation demonstrated that the OWL representation, network protocol and spatial indexing algorithm generally work well but that some improvements can be made. We provide a summary of our findings in the below subsections.

9.2.1 OWL Execution time and Memory Evaluation

We divided the evaluation into two sections: initialisation, which involved loading the ontology into the OWL reasoner and preprocessing $\exists$ statements and evaluation which measures the amount of time required to evaluate all loaded Messages.

We found that in the general case condition evaluation times were reasonable, but that initialisation times could be improved by reducing the verbosity of our OWL representation and by developing methods for restricting the range of values held by Contract parameters. Initialisation times were found to be particularly poor when many Context Value tuples were required for Message evaluation.

The required number of Context Values rose as message complexity increased, but the main contributing factor was the number of Contracts, and more importantly the number and types of their parameters. This was to be expected as the number of possible Context Values grow as the number and complexity of Contracts increase.

This is because the introduction of additional parameters increases the number of possible combinations of ContextValue tuples that will be required for evaluation. Parameters with data types such as Double and Integer are especially problematic as they offer an extremely wide range of possible values.

We performed further tests using Contracts without parameters and found that both initialisation and evaluation times improved greatly.

Memory usage was found to be a challenge, with quite simple conditions consuming over 8 GB of memory when representing 2000 Messages. This could cause scalability issues if the middleware is required to handle a large number of Messages or the Messages are distributed unevenly across the network. However, we envisage that this memory requirement could be reduced by making the representation less verbose.

When evaluating the initialisation of ∃ blocks we found that certain configurations took much longer to initialise than others. Specifically, we found that blocks which required fewer Contracts to succeed took significantly longer to initialise because the block as a whole was therefore more likely to succeed, incurring the overhead of inserting additional individuals into the ontology.

9.2.2 Simulation

The main findings of our simulations were that the network nodes tend to be unevenly distributed and that this could be problematic for the scalability of the network. In order to ensure satisfactory performance these "hotspots" should be identified and sufficient resources allocated. This could be achieved by ensuring that the available Regional nodes have enough processing power and memory or by deploying a sufficient number of nodes in the area. Fortunately, these hotspot areas should generally be quite straightforward to identify as they will tend to present themselves in highly populated areas such as cities.

The density of nodes is also an issue as if they are too sparsely distributed Participants will struggle to obtain the Context Values they require from their surrounding neighbours. We must also ensure that we achieve a good level of connectedness between Regional nodes so that Context Requests can be distributed to the closest nodes in the network.

Finally, we concluded that it was important to determine the typical condition complexity and specificity of Messages for particular use cases in order to provision the network appropriately. Ideally, we could derive some universal properties from these use cases to allow us to define a baseline specification for nodes.

9.2.3 Properties of Spatial Indexes

We found that our spatial indexing algorithm performed well in the general case, with the number of generated indexes hitting a maximum of six for all complexities of our "expected" condition class. ∃ blocks did pose a problem, as in order to represent their semantics completely sometimes requires the generation of a very large number of indexes. However, we found that these cases tended to only account for a small percentage of our data and we have proposed a throttling algorithm to restrict the number of generated indexes to a more reasonable number. The throttling algorithm does however simplify

the block which may result in more false positives during Message lookup; but importantly will not result in any false negatives.

9.2.4 System Boundary Conditions

The MediateSpace middleware is appropriate for applications where the following two properties hold:

Network Topology The MediateSpace middleware is appropriate for applications where most Participant nodes are geographically close to other Participants, allowing them to exchange ContextValue tuples when necessary. This constraint is most easily fulfilled in urban areas such as large cities and towns. However, this constraint can be relaxed when the types of context supported by the application have a wide area of applicability. For example, an accurate temperature can be shared over a larger geographical distance than location. Applications that require limited context sharing are also less bound by this constraint. For example, the Geocaching application we outline in Section 1.4.2 can be used successfully with only locally specified context.

Condition Structure Our middleware is also appropriate for applications whose Contracts require only a small amount of variability in their parameters, and ideally have short parameter lists. Application designers should also be aware of the expected specificity of conditions as the demand on nodes increases as condition specificity decreases. In cases where these properties do not hold performance can be improved by increasing the number and density of Regional nodes or by improving the hardware specifications of existing Regional nodes.

We now move on to propose potential avenues of future work.

9.3 Future Work

- Use cases could be defined for typical uses of the middleware; defining a representative condition complexity and specificity for each. This would allow us to ensure that an appropriate number of suitable Regional nodes were deployed to support the system. Universal properties could hopefully be derived from the use cases which would allow us to define a baseline specification for applications.
- Alternative Context Request protocols could be explored. For example, Fleming et al.s' [35] stigmergic approach (discussed in Section 8.4.6).
- The distribution of nodes tend to be unevenly distributed. It could be useful to run additional simulations to determine the appropriate parameters for efficiently handling the network demands of areas of vary-

ing size and node density. These parameters could then be applied to estimate the provisioning requirements for similar areas.
- Our system could be deployed on mobile devices in order to test it's effectiveness in the real world with real people.
- More restrictive data types and an explicit means of restricting parameter values for Contracts could be incorporated into the language (as discussed in Section 8.3.6.1). This would benefit the initialisation times of our OWL representation as the number of possible combinations of Context Value would be reduced.
- Optimisations for the handling of $\exists$ blocks in our OWL representation could be researched and implemented; or alternative representations could be devised.
- The memory requirements of our OWL representation could be improved by reducing its verbosity.
- The throttling algorithm we defined for the mapping of $\exists$ blocks to spatial indexes could be refined by retaining more of the semantics of the original condition, but with a comparable output.
- Security and privacy are both concerns which need to be addressed as users may be uncomfortable sharing certain types of context such as location with nearby nodes. This may be particularly true with Regional nodes as they could be managed by businesses or government agencies. Security could also be an issue as at present all messages are dispatched through the network in plain text.
- To successfully deploy our system in the real world, we would require a replication protocol to ensure fault tolerance and availability.

9.4 Summary

The evaluation confirms that our design works as intended, allowing the insertion, retrieval and evaluation of Messages using sophisticated contextual conditions capable of modelling multiple aspects of context with variable context information.

In conclusion, we achieved our aim of developing a distributed context-aware content distribution framework. This included a flexible condition and context representation language, a network topology and protocol that supports context sharing, and indexing and evaluation capabilities.

References

[1] The free on-line dictionary of computing, http://foldoc.org/, editor denis howe. URL http://foldoc.org/middleware.

[2] Gregory D. Abowd, Anind K. Dey, Peter J. Brown, Nigel Davies, Mark Smith, and Pete Steggles. Towards a better understanding of context and context-awareness. In *Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing*, HUC '99, pages 304–307, London, UK, UK, 1999. Springer-Verlag. ISBN 3-540-66550-1. URL http://dl.acm.org/citation.cfm?id=647985.743843.

[3] Jordi Pujol Ahull and Pedro Garca Lpez. Planetsim: an extensible framework for overlay network and services simulations. In Sndor Molnr, John R. Heath, Olivier Dalle, and Gabriel A. Wainer, editors, *SimuTools*, page 45. ICST, 2008. ISBN 978-963-9799-20-2. URL http://dblp.uni-trier.de/db/conf/simutools/simutools2008.html#AhulloL08.

[4] C. Archer. *Process Coordination and Ubiquitous Computing*, chapter 1, pages 11–29. CRC Press, Inc., 2002.

[5] Matthias Baldauf, Schahram Dustdar, and Florian Rosenberg. A survey on context-aware systems. *International Journal of Ad Hoc and Ubiquitous Computing*, 2(4):263–277, June 2007. ISSN 1743-8225. doi: 10.1504/IJAHUC.2007.014070. URL http://dx.doi.org/10.1504/IJAHUC.2007.014070.

[6] Albert-Laszlo Barabasi. The origin of bursts and heavy tails in human dynamics. *Nature*, 435:207, 2005. URL http://www.citebase.org/abstract?id=oai:arXiv.org:cond-mat/0505371.

[7] Albert-Lszl Barabsi. *Bursts : the hidden pattern behind everything we do*. New York, N.Y. Dutton, 2010. ISBN 978-0-525-95160-5. URL http://opac.inria.fr/record=b1130554.

[8] Mathieu Bastian, Sebastien Heymann, and Mathieu Jacomy. Gephi: An open source software for exploring and manipulating networks, 2009. URL http://www.aaai.org/ocs/index.php/ICWSM/09/paper/view/154.

[9] Christian Becker and Frank Dürr. On location models for ubiquitous computing. *Personal Ubiquitous Comput.*, 9(1):20–31, 2005. ISSN 1617-4909. doi: http://dx.doi.org/10.1007/s00779-004-0270-2.

[10] Stefan Berchtold, Daniel A. Keim, and Hans P. Kriegel. The X-Tree: An Index Structure for High-Dimensional Data. In T. M. Vijayaraman,

Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *Proceedings of the 22nd International Conference on Very Large Databases*, pages 28–39, San Francisco, U.S.A., 1996. Morgan Kaufmann Publishers. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.53.6661.

[11] Silvia Bianchi. *Load-balanced Structures for Decentralized Overlays.* PhD thesis, Instituté dInformatique, Universite De Neuchâtel, 2008.

[12] Avrim Blum. Universal and perfect hashing. University Lecture Notes, 2012. URL http://www.cs.cmu.edu/~avrim/451f12/lectures/lect0918.pdf.

[13] Elisa Gonzalez Boix, Christophe Scholliers, Wolfgang De Meuter, and Theo DHondt. Programming mobile context-aware applications with {TOTAM}. *Journal of Systems and Software*, (0):–, 2013. ISSN 0164-1212. doi: http://dx.doi.org/10.1016/j.jss.2013.07.031. URL http://www.sciencedirect.com/science/article/pii/S0164121213001799.

[14] Christian Bhm. *Efficiently Indexing High-Dimensional Data Spaces.* PhD thesis, Universitt Mnchen, 1998.

[15] D. Chalmers, M. Sloman, and N. Dulay. Map adaptation for users of mobile systems. In *Proceedings of 10th Intl. World Wide Web Conference (WWW10)*, pages 735–744. ACM, 2001.

[16] D. Chalmers, N. Dulay, and M. Sloman. Towards reasoning about context in the presence of uncertainty. In *Proceedings of Workshop on Advanced Context Modelling, Reasoning And Management at UbiComp 2004*, 2004.

[17] Guanling Chen and David Kotz. A survey of context-aware mobile computing research. Technical report, Hanover, NH, USA, 2000.

[18] Harry Chen, Tim Finin, and Anupam Joshi. An ontology for context-aware pervasive computing environments. *Knowl. Eng. Rev.*, 18(3):197–207, May 2004. ISSN 0269-8889. doi: http://dx.doi.org/DOI:10.1017/S0269888904000025.

[19] Penghe Chen, Shubhabrata Sen, Hung Keng Pung, and Wai Choong Wong. Context processing: A distributed approach. In *INTELLI 2013, The Second International Conference on Intelligent Systems and Applications*, pages 58–64, 2013.

[20] Keith Cheverst, Nigel Davies, Keith Mitchell, Adrian Friday, and Christos Efstratiou. Developing a context-aware electronic tourist guide: some issues and experiences. In *CHI '00: Proceedings of the SIGCHI conference*

on Human factors in computing systems, pages 17–24, New York, NY, USA, 2000. ACM. ISBN 1-58113-216-6. doi: 10.1145/332040.332047. URL http://dx.doi.org/10.1145/332040.332047.

[21] Chire. Visualization of an r*-tree for 3d points using elki. URL http://commons.wikimedia.org/wiki/File:RTree-Visualization-3D.svg#mediaviewer/File:RTree-Visualization-3D.svg. Accessed: 22/09/2014.

[22] Douglas Comer. Ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, June 1979. ISSN 0360-0300. doi: 10.1145/356770.356776. URL http://doi.acm.org/10.1145/356770.356776.

[23] Paolo Costa, Luca Mottola, Amy L. Murphy, and Gian P. Picco. TeenyLIME: transiently shared tuple space middleware for wireless sensor networks. In *MidSens '06: Proceedings of the international workshop on Middleware for sensor networks*, pages 43–48, New York, NY, USA, 2006. ACM Press. doi: http://doi.acm.org/10.1145/1176866.1176874. URL http://dx.doi.org/http://doi.acm.org/10.1145/1176866.1176874.

[24] Gianpaolo Cugola, Gian P. Picco, and Politecnico Di Milano. PeerWare: Core middleware support for peer-to-peer and mobile systems, 2001.

[25] Carlo Curino, Matteo Giani, Marco Giorgetta, Alessandro Giusti, Amy L. Murphy, and Gian Pietro Picco. Mobile data collection in sensor networks: The TinyLIME middleware. *Pervasive and Mobile Computing*, 1(4):446 – 469, 2005. ISSN 1574-1192. doi: DOI:10.1016/j.pmcj.2005.08.003. URL http://www.sciencedirect.com/science/article/B7MF1-4H9GRV7-1/2/bf4f596735fdf93adcfd4a0ecfd6255a. Special Issue on PerCom 2005.

[26] Frank Dabek, Ben Y. Zhao, Peter Druschel, John Kubiatowicz, and Ion Stoica. Towards a common api for structured peer-to-peer overlays. In M. Frans Kaashoek and Ion Stoica, editors, *IPTPS*, volume 2735 of *Lecture Notes in Computer Science*, pages 33–44. Springer, 2003. ISBN 3-540-40724-3. URL http://dblp.uni-trier.de/db/conf/iptps/iptps2003.html#DabekZDKS03.

[27] Amy Murphy Dept and Amy L. Murphy. LIME: A middleware for physical and logical mobility. In *Proceedings of the The 21st International Conference on Distributed Computing Systems*, ICDCS '01, pages 524–, Washington, DC, USA, 2001. IEEE Computer Society.

[28] Anind K. Dey, Gregory D. Abowd, and Daniel Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Hum.-Comput. Interact.*, 16(2):97–166, December 2001. ISSN 0737-0024. doi: 10.1207/S15327051HCI16234_02. URL http://dx.doi.org/10.1207/S15327051HCI16234_02.

[29] Antonio Di Ferdinando, Alberto Rosi, Ricardo Lent, Antonio Manzalini, and Franco Zambonelli. Myads: A system for adaptive pervasive advertisements. *Pervasive Mob. Comput.*, 5(5):385–401, October 2009. ISSN 1574-1192. doi: 10.1016/j.pmcj.2009.06.006.

[30] Nicholas Drummond, Alan Rector, Robert Stevens, Georgina Moulton, Matthew Horridge, Hai Wang, and Julian Sedenberg. Putting owl in order: Patterns for sequences in owl. In *OWL Experiences and Directions (OWLEd 2006)*, Athens Georgia, 2006.

[31] Nathan Eagle and Alex (Sandy) Pentland. Reality mining: sensing complex social systems. *Personal Ubiquitous Comput.*, 10(4):255–268, March 2006. ISSN 1617-4909. doi: 10.1007/s00779-005-0046-3.

[32] Shane B. Eisenman, Nicholas D. Lane, Emiliano Miluzzo, Ronald A. Peterson, Gahng seop Ahn, and Andrew T. Campbell. Metrosense project: People-centric sensing at scale. In *WSW 2006 at Sensys*, 2006.

[33] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003. ISSN 0360-0300. doi: 10.1145/857076.857078. URL http://doi.acm.org/10.1145/857076.857078.

[34] Patrick Fahy and Siobhan Clarke. Cass a middleware for mobile context-aware applications. In *Workshop on Context Awareness, MobiSys*, 2004.

[35] Simon Fleming, Dan Chalmers, and Ian Wakeman. A deniable and efficient question and answer service over ad hoc social networks. *Information Retrieval*, 15(3-4):296–331, 2012. ISSN 1386-4564. doi: 10.1007/s10791-012-9185-0. URL http://dx.doi.org/10.1007/s10791-012-9185-0.

[36] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2.

[37] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesc language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03,

pages 1–11, New York, NY, USA, 2003. ACM. ISBN 1-58113-662-5. doi: 10.1145/781131.781133. URL http://doi.acm.org/10.1145/781131.781133.

[38] David Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, January 1985. ISSN 0164-0925. doi: 10.1145/2363.2433. URL http://doi.acm.org/10.1145/2363.2433.

[39] David Gelernter. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, January 1985. ISSN 0164-0925. doi: 10.1145/2363.2433. URL http://dx.doi.org/10.1145/2363.2433.

[40] Hans W. Gellersen, Albercht Schmidt, and Michael Beigl. Multi-sensor context-awareness in mobile devices and smart artifacts. *Mob. Netw. Appl.*, 7(5):341–351, October 2002. ISSN 1383-469X. doi: 10.1023/A:1016587515822. URL http://dx.doi.org/10.1023/A:1016587515822.

[41] Larry Gonick and Woollcott Smith. *The Cartoon Guide to Statistics*. HarperResource, February 1994. ISBN 0062731025. URL http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0062731025.

[42] Marta C. Gonzalez, Cesar A. Hidalgo, and Albert-Laszlo Barabasi. Understanding individual human mobility patterns. *Nature*, 453(7196):779–782, June 2008. doi: 10.1038/nature06958.

[43] Michael T. Goodrich and Roberto Tamassia. *Algorithm design - foundations, analysis and internet examples.* Wiley, 2002. ISBN 978-0-471-38365-9.

[44] William Grosso. *Java RMI.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1st edition, 2001. ISBN 1565924525.

[45] Tao Gu, Hung K. Pung, and Da Q. Zhang. A service-oriented middleware for building context-aware services. *Journal of Network and Computer Applications*, 28(1):1–18, January 2005. doi: 10.1016/j.jnca.2004.06.002.

[46] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, volume 14, pages 47–57, New York, NY, USA, June 1984. ACM. ISBN 0-89791-128-8. doi: 10.1145/602259.602266. URL http://dx.doi.org/10.1145/602259.602266.

[47] Hamed Haddadi, Pan Hui, Tristan Henderson, and Ian Brown. *Targeted Advertising on the Handset: Privacy and Security Challenges*. Human-Computer Interaction Series. Springer, July 2011.

[48] P.D. Haghighi, A Zaslavsky, and S. Krishnaswamy. An evaluation of query languages for context-aware computing. In *Database and Expert Systems Applications, 2006. DEXA '06. 17th International Workshop on*, pages 455–462, 2006. doi: 10.1109/DEXA.2006.25.

[49] Karen Henricksen, Jadwiga Indulska, and Andry Rakotonirainy. Generating context management infrastructure from high-level context models. In *In 4th International Conference on Mobile Data Management (MDM) - Industrial Track*, pages 1–6, 2003.

[50] Thomas Hofer, Wieland Schwinger, Mario Pichler, Gerhard Leonhartsberger, Josef Altmann, and Werner Retschitzegger. Context-awareness on mobile devices - the Hydrogen approach. In *HICSS '03: Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03) - Track 9*, page 292.1, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1874-5.

[51] Matthew Horridge and Sean Bechhofer. The owl api: A java api for owl ontologies. *Semant. web*, 2(1):11–21, January 2011. ISSN 1570-0844. URL http://dl.acm.org/citation.cfm?id=2019470.2019471.

[52] Matthew Horridge, Nick Drummond, John Goodwin, Alan Rector, Robert Stevens, and Hai Wang. The manchester owl syntax. In *OWLED2006 Second Workshop on OWL Experiences and Directions*, Athens, GA, USA, 2006.

[53] Ian Horrocks, Peter F. Patel-Schneider, and Frank Van Harmelen. From shiq and rdf to owl: The making of a web ontology language. *Journal of Web Semantics*, 1:2003, 2003.

[54] Ibrahim Kamel and Christos Faloutsos. Hilbert r-tree: An improved r-tree using fractals. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 500–509, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc. ISBN 1-55860-153-8. URL http://dl.acm.org/citation.cfm?id=645920.673001.

[55] Panu Korpipaa, Jani Mantyjarvi, Juha Kela, Heikki Keranen, and Esko-Juhani Malm. Managing context information in mobile devices. *IEEE Pervasive Computing*, 2(3):42–51, 2003. ISSN 1536-1268.

[56] King I. Lin, H. V. Jagadish, and Christos Faloutsos. The TV-Tree: An Index Structure for High-Dimensional Data. *VLDB Journal: Very Large Data Bases*, 3(4):517–542, 1994. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.57.3060.

[57] David Malan, Thaddeus Fulford-Jones, Matt Welsh, and Steve Moulton. Codeblue: An ad hoc sensor network infrastructure for emergency medical care. In *In International Workshop on Wearable and Implantable Body Sensor Networks*, 2004. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.113.7341.

[58] Yannis Manolopoulos, Alexandros Nanopoulos, Apostolos N. Papadopoulos, and Y. Theodoridis. *R-Trees: Theory and Applications*, chapter 4, pages 55–61. Springer, 1 edition, September 2005.

[59] Danny Matthews, Dan Chalmers, and Ian Wakeman. Mediatespace: decentralised contextual mediation using tuple spaces. In *Proceedings of the Third International Workshop on Middleware for Pervasive Mobile and Embedded Computing*, M-MPAC '11, pages 5:1–5:8, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1065-9. doi: 10.1145/2090316.2090321.

[60] Danny Matthews, Dan Chalmers, and Ian Wakeman. Improving the effectiveness of advertising through contextual mediation. 2012. URL http://www.dmatthews.co.uk/resources/publications/pervasive-advertising-2012-paper.pdf.

[61] M. Mauve, A. Widmer, and H. Hartenstein. A survey on position-based routing in mobile ad hoc networks. *Netwrk. Mag. of Global Internetwkg.*, 15(6):30–39, November 2001. ISSN 0890-8044. doi: 10.1109/65.967595. URL http://dx.doi.org/10.1109/65.967595.

[62] Ted McFadden, Karen Henricksen, and Jadwiga Indulska. Automating context-aware application development. In *In: UbiComp 1st International Workshop on Advanced Context Modelling, Reasoning and Management*, pages 90–95, 2004.

[63] Robin Milner. Bigraphs and their algebra. *Electronic Notes in Theoretical Computer Science*, 209:5–19, April 2008. ISSN 15710661. doi: 10.1016/j.entcs.2008.04.002. URL http://dx.doi.org/10.1016/j.entcs.2008.04.002.

[64] B Moltchanov, M Knappmeyer, C.A. Licciardi, and N Baker. Context-aware content sharing and casting. In *Proceedings of ICIN 2008, Bordeaux, France*, 2008.

[65] M. E. J. Newman. Power laws, pareto distributions and zipf's law. *Contemporary Physics*, 46:323–351, December 2005. URL http://arxiv.org/abs/cond-mat/0412004.

[66] Gian Pietro Picco, Davide Balzarotti, and Paolo Costa. LighTS: A Lightweight, Customizable Tuple Space Supporting Context-Aware Applications. In *Proceedings of the 20th ACM Symposium on Applied Com-*

puting (SAC05), SAC 05, pages 1134–1140, Santa Fe (New Mexico, USA), March 2005. ACM Press.

[67] Stavros Polyviou, Paraskevas Evripidou, and George Samaras. Contextaware queries using query by browsing and chiromancer. In *Second International Conference on Pervasive Computing*, 2004.

[68] Julien Ponge. Avoiding benchmarking pitfalls on the jvm. *Java Magazine*, pages 42–50, 2014. URL http://www.oracle.com/javamagazine.

[69] M. Porter. The Porter Stemming Algorithm. URL http://www.tartarus.org/martin/PorterStemmer.

[70] Q-Success. Usage of advertising networks for websites. URL http://w3techs.com/technologies/overview/advertising/all. Accessed: 22/09/2014.

[71] Anand Ranganathan and Roy H. Campbell. An infrastructure for context-awareness based on first order logic. *Personal Ubiquitous Comput.*, 7(6):353–364, December 2003. ISSN 1617-4909. doi: 10.1007/s00779-003-0251-x. URL http://dx.doi.org/10.1007/s00779-003-0251-x.

[72] Rajiv Ranjan, Aaron Harwood, and Rajkumar Buyya. Peer-to-Peer Tuple Space: A novel protocol for coordinated resource provisioning. Technical report, The University of Melbourne, Victoria, Australia, 2007.

[73] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 161–172, New York, NY, USA, 2001. ACM. ISBN 1-58113-411-8. doi: 10.1145/383059.383072. URL http://doi.acm.org/10.1145/383059.383072.

[74] Simon Rogers. Uk population: find out what's happened near you. URL http://www.theguardian.com/news/datablog/2011/jun/30/uk-population-growth-data. Accessed: 22/09/2014.

[75] Antony Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. Scribe: The design of a large-scale event notification infrastructure. In *In Networked Group Communication*, pages 30–43, 2001.

[76] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Middleware '01, pages 329–350, London,

UK, UK, 2001. Springer-Verlag. ISBN 3-540-42800-3. URL http://dl.acm.org/citation.cfm?id=646591.697650.

[77] Hans. Sagan. *Space-Filling Curves*, chapter 2, pages 9–30. Springer-Verlag New York, Inc., 1994.

[78] Daniel Salber, Anind K. Dey, and Gregory D. Abowd. The Context Toolkit: aiding the development of context-enabled applications. In *CHI '99: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 434–441, New York, NY, USA, 1999. ACM. ISBN 0-201-48559-1. doi: http://doi.acm.org/10.1145/302979.303126.

[79] J. Santa and A. F. Gómez-Skarmeta. Sharing context-aware road and safety information. *Pervasive Computing, IEEE*, 8(3):58–65, July 2009. ISSN 1536-1268. doi: 10.1109/MPRV.2009.56.

[80] Katie Shilton. Four billion little brothers?: privacy, mobile phones, and ubiquitous data collection. *Commun. ACM*, 52(11):48–53, 2009. ISSN 0001-0782. doi: 10.1145/1592761.1592778. URL http://dx.doi.org/10.1145/1592761.1592778.

[81] Yan Shvartzshnaider, Maximilian Ott, and David Levy. Publish/-subscribe on top of dht using rete algorithm. In *Proceedings of the Third Future Internet Conference on Future Internet*, FIS'10, pages 20–29, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-15876-5, 978-3-642-15876-6. URL http://dl.acm.org/citation.cfm?id=1929268.1929271.

[82] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, volume 31, pages 149–160, New York, NY, USA, October 2001. ACM. ISBN 1-58113-411-8. doi: 10.1145/383059.383071. URL http://dx.doi.org/10.1145/383059.383071.

[83] Thomas Strang and Claudia L. Popien. A context modeling survey. In *Workshop on Advanced Context Modelling, Reasoning and Management, UbiComp 2004 - The Sixth International Conference on Ubiquitous Computing, Nottingham/England*, September 2004. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.2.2060.

[84] Theory.org. Bittorrent protocol specification v1.0. http://wiki.theory.org/BitTorrentSpecification, September 2014.

[85] D. Tsarkov and I. Horrocks. Fact++ description logic reasoner: System description. In *Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2006)*, volume 4130 of *Lecture Notes in Artificial Intelligence*, pages 292–297. Springer, 2006.

[86] Dmitry Tsarkov and Ian Horrocks. Fact++ description logic reasoner: System description. In *Proceedings of the Third International Joint Conference on Automated Reasoning*, IJCAR'06, pages 292–297, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-37187-7, 978-3-540-37187-8. doi: 10.1007/11814771_26. URL http://dx.doi.org/10.1007/11814771_26.

[87] Mathieu Valero, Luciana Arantes, Maria Potop-Butucaru, and Pierre Sens. Enhancing fault tolerance of distributed r-tree. In *LADC*, pages 25–34. IEEE Computer Society, 2011. URL http://dblp.uni-trier.de/db/conf/ladc/ladc2011.html#ValeroAPS11.

[88] Xiang-Wen Wang, Xiao-Pu Han, and Bing-Hong Wang. Correlations and scaling laws in human mobility. *PLoS ONE*, 9(1):e84954, 01 2014. doi: 10.1371/journal.pone.0084954.

[89] Roy Want, Andy Hopper, Veronica Falcão, and Jonathan Gibbons. The active badge location system. *ACM Trans. Inf. Syst.*, 10(1):91–102, January 1992. ISSN 1046-8188. doi: 10.1145/128756.128759. URL http://doi.acm.org/10.1145/128756.128759.

[90] Mark Weiser. Some computer science issues in ubiquitous computing. *Commun. ACM*, 36(7):75–84, July 1993. ISSN 0001-0782. doi: 10.1145/159544.159617. URL http://doi.acm.org/10.1145/159544.159617.

[91] Mark Williams. Annual survey of hours and earnings, 2010 revised results. URL http://www.ons.gov.uk/ons/rel/ashe/annual-survey-of-hours-and-earnings/2010-revised-results/index.html. Accessed: 22/09/2014.

[92] Derick Wood. *Data structures, algorithms, and performance.* Addison-Wesley, 1993. ISBN 978-0-201-52148-1.

[93] Zhichen Xu, Chunqiang Tang, and Zheng Zhang. Building topology-aware overlays using global soft-state. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, ICDCS '03, pages 500–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1920-2. URL http://dl.acm.org/citation.cfm?id=850929.851967.