UNIVERSITY OF SOUTHAMPTON

Chained Negotiation for Quality of Service in Distributed Notification Services

by

Richard A. Lawley

A thesis submitted in partial fulfillment for the degree of Doctor of Philosophy

in the
Faculty of Engineering and Applied Science
Department of Electronics and Computer Science

September 2005

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING AND APPLIED SCIENCE
DEPARTMENT OF ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

by Richard A. Lawley

With the growth of the Internet over recent years, the use of distributed systems has increased dramatically. Components of distributed systems require a communications infrastructure in order to interact with other components. One such method of communication is a notification service (NS), which delivers notifications of events between publishers and consumers that have subscribed to these events. A distributed NS is made up of multiple NS instances, enabling publishers and consumers to be connected to different NSs and still communicate. The NSs attempt to optimise message flow between them by sharing subscriptions between consumers with similar interests. In many cases, there is a mismatch between the dissemination notifications from a publisher and the delivery preferences of the consumer in terms of frequency of delivery, quality, etc. Consumers wish to receive a high quality of service, while a service provider acting as a publisher wishes to make its service available to many consumers without overloading itself. Negotiation is applicable to the resolution of this mismatch. However, existing forms of negotiation are incompatible with distributed NSs, where negotiation needs to take into account the preferences of the publisher and consumer, as well as existing subscriptions held by NSs. We introduce the concept of chained negotiation, where one or more intermediaries sit between the client and supplier in a negotiation, as a solution to this problem. Automated chained negotiation can enable a publisher and consumer to find a mutually acceptable set of delivery preferences for a service to be delivered through a distributed NS, while still enabling NSs to share subscriptions between consumers with similar interests. In this thesis, we present the following contributions: first, we show that by using negotiation over quality of service conditions, a service provider can serve more clients with a lower load on itself, presenting a direct negotiation engine for this purpose. We present chained negotiation as a novel form of negotiation enabling quality of service negotiations to involve intermediaries which may be able to satisfy a client's request without involving the service provider. Finally, we present a distributed notification service with support for chained negotiation, showing the benefit gained from chained quality of service negotiation in a real application.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my two supervisors Professor Luc Moreau and Professor Mike Luck for their guidance throughout the period of my research, without which this thesis would not have been possible. I have also received helpful input from Simon Miles and Paul Groth, who both helped me with the experiment used in Chapter 6.

I would also like to thank my friends, parents and colleagues for all of their help and support over the last few years.

This research has been funded in part by EPSRC myGrid project (ref. GR/R67743/01).

Chapter 1

Introduction

1.1 Distributed Systems

Possibly the biggest change in the field of computing over the past decade has been the explosive growth of the Internet, linking networks all over the world and enabling people to communicate and collaborate, sharing information. It has also enabled computer systems to interact with other geographically distributed computer systems, almost as easily as if they were locally connected. Stock control systems in shops can automatically place orders with their suppliers where previously a human would have placed an order by telephone or mail, customers can track the current location of a parcel they have shipped anywhere in the world, and traders can monitor and trade stocks and shares in real time where they would previously have telephoned a broker. Prior to the Internet, businesses could interact electronically by means of private dial-up networks, which were costly, slow and restricted in terms of which organisations they could interact with. The Internet has provided an affordable global network which enables interaction with millions of other organisations around the world.

While the most common use of the Internet is to enable *people* to communicate with other people, or to interact with businesses and other organisations electronically, a more recent development has seen improvements in methods enabling *machines* to interact with each other — *Web Services*. A Web Service is a software system designed to support interoperable machine-to-machine interaction over a network (Web Services Architecture Working Group, 2004). Web services encompass a universal language for the exchange of data between applications and protocols for remotely discovering and accessing electronic services with a machine-processable interface, and enable applications to interact with other applications distributed anywhere in the world.

1.1.1 The Grid

Access to a global network has also enabled scientific projects to collaborate, leading to the vision of the Grid, which is a system enabling the distributed coordination of resources. These resources can include computing power, storage space, databases, scientific apparatus and any other service or device that can be networked. Having these shared resources allows a group of people who have never met to dynamically form a *virtual organisation* in order to collaborate on a task (Norman et al., 2004). When the task has completed, the virtual organisation may be disbanded without the individuals involved ever knowing their collaborators. While *the* Grid is still a vision, many grid systems have been created, realising some of the ideas in the Grid concept. Scientific applications are major users of grid systems — they are typically computation- or data-intensive, making them ideal candidates for using shared resources and making their own resources available. By sharing resource and using shared resources, an organisation can typically run such an application at a lower cost than by having to acquire and support the necessary resources themselves.

1.1.2 Service-Oriented Architectures and Quality of Service

Both Web Services and grid systems have led to the notion of a *service-oriented architecture* (Burbeck, 2000) — an architecture that focuses on the description of services and supporting their dynamic, automated discovery and use. Services offered can vary in complexity from simple stock quote services to more complex scientific experiments or database searches. *Service providers* may exist for the sole purpose of providing services to others, potentially for financial reward.

In some situations, it may be insufficient to simply request a service make assumptions about the quality of service. For example, a scientific experiment may have some equipment time preallocated in the future, but requires an external service to perform some analysis on existing data before it can use the equipment. If the external service does not complete in time, the pre-booked equipment time is wasted. Hence, it is often desirable to specify constraints about *how* a service is delivered, which we broadly refer to as *Quality of Service* (QoS). QoS parameters control how a service is delivered, as opposed to what service is delivered or the inputs to that service, e.g. service delivery time, or accuracy with which a service is run.

There is commonly a difference between the levels of QoS a client would like to get, and those which a service provider would like to provide. Service providers may try and provide lower QoS than a client would prefer to receive in order to reduce their costs or increase their throughput, enabling them to serve more clients. A small number of clients requesting a very high QoS could overload a service provider, making it unavailable to subsequent clients. In such situations, negotiation can be used to find a compromise

between the QoS levels a client prefers and those the provider is willing to supply, as is commonly used in distributed real-time systems to specify requirements for resources (Li and Ravindran, 2004), and in multimedia systems for resource reservation (Rothermel et al., 1997).

1.1.3 Notification Services

Computationally-intensive experiments in grid systems make take hours, days or even weeks for each stage, composed of a grid service made available for use, to complete. It is undesirable for a service to be waiting on a previous stage of the experiment to complete, as the time spent waiting could be better used to serve other experiments. In such situations, a message-based system where actions can be triggered on receipt of an event notification may be used to trigger the next part of the experiment, freeing up the waiting services to carry out other actions while they are free. Notification Services (NSs) are message-based communication systems that are used to inform *consumers* that an event has taken place, by delivering a notification to them. These notifications are created by *publishers*, which are information sources, and are published on a specific topic, grouping similar messages together. Consumers subscribe to the topic they are interested in, and then receive all notifications published on that topic. A NS could be used to inform users that one stage of an experiment has been completed, or that a stock price they are monitoring has changed. It is possible for a consumer to request specific QoS levels when they subscribe to a topic. For example, they could request that their notifications are compressed, or specify a minimum interval between notifications.

A common trait with any distributed system is that as the use of a system scales up, one part can become a bottleneck or a potential single point of failure for the entire system. A NS could become such a bottleneck in a distributed system, as it can potentially support many different services and applications. Having multiple NSs can enable a system to stay running if one fails, and also means that publishers and consumers can be spread between the different instances, spreading the load. Hence, an extension of a notification service is a *distributed* notification service, made up of multiple individual notification services linked together (Krishna et al., 2003). A distributed NS can also enable a wider range of users to be serviced — users behind a firewall could be connected to a local NS that can connect through the firewall to other parts of the distributed NS. In a distributed NS, messages are still delivered to consumers if the publisher is connected to a different NS. The routing of messages between the various NSs is handled by the NSs; the publisher and consumer do not even need to know they are connected by a distributed NS. When multiple consumers at the same NS are subscribing to the same topic published at a different NS, an optimisation can be made in the message routing. Instead of an individual copy of every notification being sent between the NSs, a single copy could be sent, then redistributed to each consumer by the NS. The consumers' NS

only needs to make a single subscription to the publishing NS, which it will then share between the subscribed consumers. Hence, we refer to *shared subscriptions* as those made by a NS to another NS in order to share notifications between multiple consumers interested in the same topic.

However, as mentioned above, consumers can request specific QoS conditions along with a subscription. For example, notifications can be assigned a priority to ensure that urgent notifications are delivered quickly, or a particular format of notification can be specified. For a subscription to be shared, the topic has to be the same, and the QoS conditions have to be *compatible* — whether a particular QoS can satisfy another QoS request. For example, if price is a QoS condition, and a request specifies a certain price, if the service is obtained for less, that is compatible with the QoS request. In this example, QoS conditions *better* than requested are compatible with the request. Other QoS conditions may only be considered compatible if the QoS value is exactly as requested. If two consumers request a subscription to the same topic, but their QoS conditions are incompatible, it would be impossible for the NS to share a single subscription between them; instead, it would need to make two individual subscriptions to the same topic, one for each consumer. As the scale of this problem increases beyond two consumers, it could potentially lead to a large number of additional subscriptions being required, greatly increasing the number of notifications that must be sent between NSs and increasing the overall load on the system. To resolve this, a mechanism is required that can find a compromise between the QoS preferences of a consumer, the QoS preferences of a publisher and any existing subscriptions already held by a NS, in order to maximise the benefits of sharing subscriptions and notifications.

1.2 Research Aims

The problem we are seeking to address in this thesis is to allow a service provider to support more consumers, yet minimise system load by enabling a compromise to be reached between the levels of QoS a consumer would like and manageable levels a service provider can maintain for many consumers without placing too much load on itself. We believe that adopting work done on negotiation in the field of agent-based computing provides a means to improve the performance of such systems. Our research aims can therefore be enumerated as follows:

1. As consumers and service providers typically have different preferences about QoS when requesting a service, a mechanism is required to manage the provision of service or notifications that can find a compromise between the high QoS levels requested by the consumer, and the manageable QoS levels desired by the service provider. One technique which can be used for this purpose is automated

negotiation. We aim to show that direct negotiation is suitable for resolving these differences, and by empirical evaluation show the behaviour of such a system.

The contributions provided by completing this aim are an empirical evaluation of a direct negotiation engine showing the behaviour of such a system independent of external influences, and a demonstration that negotiation is a suitable mechanism for enabling a service provider to allow a consumer to specify QoS conditions when requesting a service without imposing an unmanageable load on the service provider, restricting the number of clients it can support.

2. As the scale of a NS deployment increases, a distributed NS is often used instead. In this situation, multiple NSs are interconnected, with publishers and consumers spread between them. Where multiple consumers at the same NS subscribe to the same topic published at a remote NS, the local NS can share a single subscription to the remote NS, sharing notifications received on that topic between the subscribed consumers. However, if the consumers request different QoS constraints it may be impossible to share a subscription between them. In this situation, negotiation offers an appropriate means to find a compromise between the QoS levels requested by the consumer, those desired by the service provider, and any existing subscriptions held by a local NS. However, direct negotiation is unable to provide a solution in this case, as it only involves two parties, the consumer and service provider. We thus need to develop a new form negotiation capable of involving intermediaries between a service provider and a consumer. In this situation, a *chain* is formed between the service provider and consumer, with NSs as intermediate steps. In consequence, we aim to develop a new form of negotiation that will involve intermediaries in the chain. This *chained negotiation* method will enable a service provider to support more consumers more efficiently than by allowing them to obtain their requested QoS levels.

   Chained negotiation, being a new form of negotiation, will represent a contribution of this thesis, enabling the reselling or redistribution of items obtained through negotiation.

3. Although many existing NSs allow a consumer to specify QoS constraints when subscribing to notifications, none currently use negotiation to find a set of QoS conditions acceptable to both the consumer and service provider publishing the notifications. We thus propose to develop a new architecture based on a distributed NS that will enable multiple consumers to request levels of QoS for a service that will be delivered as notifications through a NS. Our objective is to be able to find a compromise between the typically high preferences of the consumer, the manageable levels of QoS favoured by a service provider, and existing subscriptions to the service held by intermediaries between the consumer and service provider. This will enable the load on the service provider to be kept to a manageable level, while enabling the distributed NS to efficiently share subscriptions between

consumers with similar preferences. To show the validity of this approach, we propose to take a scientific application from the bioinformatics field and develop it to work within this architecture, showing any improvements made by using our approach.

This new architecture enabling QoS negotiation for services in a distributed notification service will represent a contribution of this thesis, along with the evaluation of the architecture demonstrating the benefits of using it in a real application.

1.3 Thesis Structure

The remainder of this thesis is organised as follows:

- In Chapter 2 we introduce notification services, explaining the need for them and reviewing existing implementations.
- In Chapter 3, we discuss negotiation. Different negotiation mechanisms are introduced before examining automated negotiation. We review existing automated negotiation systems.
- In Chapter 4 we take a bilateral negotiation model and develop *DiNE*, a *direct negotiation engine*. Through simulations, we show that using negotiation over QoS enables a service provider to offer services at a QoS level acceptable to both service provider and client, enabling more clients to be served.
- In Chapter 5, we develop the negotiation engine discussed in Chapter 4 to support *chained negotiation*, where one or more intermediaries are involved in the negotiation between client and service provider. We present a negotiation model supporting chained negotiation, which is developed into *ChaNE*, a chained negotiation engine. We show through simulations that in the context of a distributed notification service, chained negotiation can enable consumers and publishers to negotiate over QoS levels, while still being able to share subscriptions to notifications between consumers with similar preferences.
- In Chapter 6 we integrate an existing distributed notification service, and integrate it with our chained negotiation engine, creating an architecture for delivering services with negotiable quality of service through a distributed notification service. We adapt a specific example of a protein compressibility analysis application to make use of the NS, enabling us to show that a service provider can reduce the load on itself and support more clients by using chained negotiation with a distributed NS.
- Finally, we present our conclusions in Chapter 7 and discuss future work.

Chapter 2

Notification Services

In a distributed system where services depend on other service running in different locations, messages can be delivered by *notification services* informing an object that an event has taken place. In large scale distributed systems, these notification services can make delivery of these messages more efficient amongst large numbers of recipients. In this chapter, we explain the concepts behind notification services, and review existing work in the field.

2.1 Introduction

With the increase of the Internet and global networks over the past few years, the use of distributed computer systems has grown significantly. This has led to the vision of the Grid — a system that "coordinates resources that are not subject to centralised control, using standard, open, general-purpose protocols and interfaces to deliver non-trivial qualities of service" (Foster, 2002). In this vision, teams of scientists or users around the world can dynamically form a group or *virtual organisation* in order to collaborate, sharing heterogeneous services. After the task has been completed, the virtual organisation may disband without the members ever having known their collaborators. At the moment, some of the vision of the Grid remains just that. *The* Grid requires open standards and protocols to be agreed on and used such that resources can be shared and used by *any* interested party, not just one from the same system. However, parts of the Grid vision have been realised by many different grid systems. These are typically used to support scientific applications that are computationally-intensive, data-intensive and/or resource-intensive. They provide mechanisms for the discovery and accessing remote services dynamically at runtime. e-Science applications are major uses of grid systems, utilising them to carry out experiments in the areas of bioinformatics (myGrid Project, 2003), combinatorial chemistry (Comb-e-chem Project, 2003), and physics (GriPhyN Project, 2005) for example. Experiments can be carried out *in silico*, or can make use

of networked laboratory equipment exposed as a grid resource.

At the same time, new uses for the Internet have taken shape. While the Internet has been traditionally used for enabling interactions between humans, providing sites full of information and enabling communication between people, very little had been done until recently to make these features universally available to machines. *Web Services* are software systems enabling machine-to-machine interactions, encompassing techniques such as data interchange (using XML-based protocols), service discovery (UDDI), machine-readable descriptions of a service (Christensen et al., 2001) and remote invocation of services (World Wide Web Consortium, 2003). As web services have matured and gained in popularity, grid systems have begun to adopt web services techniques. For example, one of the most recent incarnations of a grid framework is the Open Grid Services Infrastructure, which is based on web services technologies.

Both grid systems and web services are examples of a *service oriented architecture* (SOA), which is a model exposing functional units of an application through *services* (Colan, 2004). Services have well-defined interfaces, and may be accessed in a platform-independent manner, enabling services on different operating systems and hardware to interact. Different services can be composed together, and made available as a service itself. SOAs have been around for a long time in one form or another — CORBA and DCOM are both older examples. The concept of a SOA today typically (but not exclusively) includes the use of XML for data interchange and WSDL for describing the interfaces of a service.

Experiments in grid systems, and complex services in SOAs, may be expressed as *workflows*, structures composing different services or components together to accomplish specific goals. In computationally-intensive experiments, each stage of a workflow may take a long time to complete. For this reason, it is undesirable to have a service or resources tied up waiting for a previous stage of the workflow to complete. Instead, a messaging-based model can be used to send the output of one stage of a workflow onto the next stage(s). A *Message-Oriented Middleware* (MOM) provides the basis for such an architecture by facilitating messaging between different components or services. There are many variations of MOMs, including message queueing systems and *notification services.*

A notification service (NS) is a messaging system that delivers notifications of events to consumers who have registered an interest in receiving them. The notifications come from publishers, which are information sources. NSs enable a publisher to distribute information to many consumers without needing to be aware that they even exist.

For the remainder of this chapter, we review message-oriented middleware systems, specifically notification services, as communication mechanisms for a distributed system. In Section 2.2, we examine two communication patterns which are relevant to this chapter. Then in Section 2.3, we introduce message-oriented middleware systems as a

mechanism for remote invocation of services and interaction between distributed components. We move on to Notification Services in Section 2.4, then discuss distributed notification services in Section 2.4.2, which increase the scalability and reliability of a notification service. Section 2.5 discusses standardisation efforts for notification services, and Section 2.6 reviews some existing notification services. We summarise in Section 2.7 and discuss how this is relevant to our aims in Section 2.8.

2.2 Communication Patterns

In software development today, there are many different types of communication patterns between parts of a program. Some of these patterns are also used for communication between components in a distributed system. We describe some of these communication patterns below, so that we may draw on their definitions later in this chapter.

2.2.1 Remote Procedure Calls

Remote Procedure Calls (RPCs) are a paradigm for providing communication between programs over a network (Birrell and Nelson, 1984), and are based on the simple model of procedure calls within a program where control is transferred to another portion of the same program on the same computer. RPC simply extends this paradigm to allow control and data to be transferred to a procedure running in another program, usually on another computer in a network. As shown in Figure 2.1, when a remote procedure call is executed, the calling environment is suspended and state information is passed to the *callee*, or the environment where the remote procedure will be executed. When this procedure finishes, the state information including the procedure result is passed back to the *caller*, and execution continues.

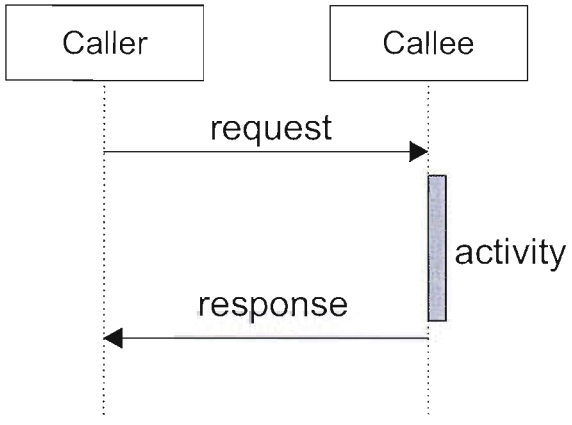

FIGURE 2.1: Sequence Diagram of RPC Interactions

For the caller, an RPC interaction is *synchronous*, meaning that the caller must stop and wait for the remote activity to complete before continuing its execution. This could

be a problem in situations where the remote activity is going to take a long time, or when the communication link between the two is unreliable.

Most of today's high-level programming languages support RPC, either through an OS implementation or as part of the language, such as Java RMI (Microsystems, 2003), Microsoft's DCOM (Horstmann and Kirtland, 1997), and SOAP (World Wide Web Consortium, 2003), which provides a mechanism for achieving RPC-style interactions in a web services environment, in addition to a much wider range of interaction styles.

2.2.2 Publish/Subscribe Interactions

In object-oriented software engineering, the *observer* interaction pattern is used where there is a dependency between a *subject*, and a number of *observers* — objects interested in changes to the state of the subject (Gamma et al., 1995). A subject can have any number of observers, as it does not need to be aware of them. Whenever a change occurs in the state of the subject, it notifies all observers of this change, using a mechanism that does not change whether it is notifying 0, 1 or 1000 observers of the changes. Figure 2.2 shows the interactions between the subject and observers — observers subscribe to a subject in order to receive notifications of updates. When the subject is updated, it notifies any observers of these changes.

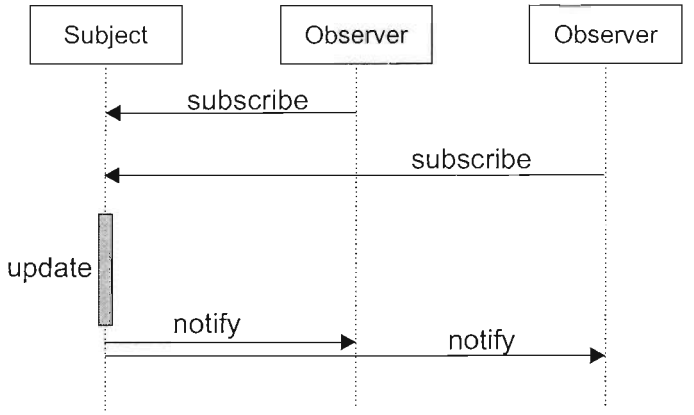

FIGURE 2.2: Sequence Diagram of Publish/Subscribe Interactions

The observer pattern is used in situations where changes in one subject cause events to occur in an indeterminate number of observers, without the subject needing to interact directly with the observers. In software development, it is often used to enable a graphical user interface to display updates to underlying data structures. It is also used in distributed systems, where it is more commonly known as *publish/subscribe* (Birman, 1993). In publish/subscribe Subjects are *publishers* of notifications, and any number of *consumers* (observers) can subscribe to receive these notifications. Publish/subscribe resembles multicast in nature, as it allows a publisher to reach multiple consumers with no difference from sending the message to a single consumer.

2.3 Message-Oriented Middleware

Over recent years, there has been a significant increase in the number of computers and other devices wishing to access and run remote services over a network. Traditionally, this would have been accomplished using RPC (as described above), but there are a number of disadvantages to this approach. RPC usually requires a client to issue their request, then await a response. This is often undesirable or difficult to achieve. For example, a user can invoke a long-running job from a PDA while on the move. It is impractical for them to stay connected while the service runs, as the device has limited battery life and an unreliable or expensive network connection. Additionally, RPC interactions generally mean a service is run immediately, which makes it difficult for a service provider to handle load from multiple clients simultaneously.

An alternative method of distributed interaction is distributed messaging, where communications are based on the exchange of messages (often referred to as events, as messages are often sent to indicate that something has taken place). *Message-oriented middleware* (MOM) systems are messaging systems facilitating the asynchronous, reliable communication between entities by the exchange of messages, and can be used for a range of scenarios, including integrating distributed applications (Banavar et al., 1999b), information exchange and event notification. They are an extension to the client-server paradigm of computing to enable asynchronous operations, so that a client can send a request to a server that is currently off-line, or a server can operate without the worry that a client is waiting for an immediate response (Birman, 1996). They also enable clients to communicate with servers without ever being directly connected — the MOM handles routing of messages as appropriate. Essentially, MOM systems are similar to an e-mail system for applications, letting different programs use named *mailboxes* for sending and receiving, and acting upon the contents of the messages (Birman, 1996). A subset of MOM, Message Queuing Middleware (MQM), uses queues for sending and receiving messages, further decoupling the client or server from the MOM by making the send or receive asynchronous (as compared to synchronous, where they have to wait for the message to be sent or for the reply to be received). By enabling simultaneous access to both sending and receiving queues, a MQM system can support many users. MQMs also facilitate load balancing by using queues — multiple service providers could be connected to a queue of requests, sharing the load between them without their clients knowledge.

MOM systems solve some of the problems associated with a traditional RPC-based approach: long-running operations can be scheduled without the client having to wait on-line for a response; service providers can handle a large number of simultaneous clients, potentially by transparently sharing the load with other service providers. MOM also enables a simpler programming model than RPC — it is possible for a client to make use of a remote service without specifying where or which server should carry out the

request. Instead, the client locates a messaging server and sends the request. The client does not need to know which service carries out the request, or what software it is running.

MOM systems have been around for some time, hence there are many established implementations available. Examples include Microsoft Message Queue (MSMQ) [1], IBM's Websphere MQ[2] (formerly MQSeries), and DEC's MessageQ, which allow reliable asynchronous communications within guaranteed delivery constraints.

2.4 Notification Services

A *notification service* (NS) is part of a MOM utilising the *Publish/Subscribe* model described above. In the above description of publish/subscribe, no mention is made of how notifications reach the consumers from the publisher. A NS is an object that takes this responsibility, taking notifications from the publisher and handling the distribution and delivery to the subscribed consumers (Hapner et al., 2002; Object Management Group, 2004, 2002). A NS uses the publish/subscribe model in two locations as shown in Figure 2.3 — to receive notifications from the publisher (the publisher notifies the NS of any notifications), and to publish these notifications to the consumers (consumers subscribe to the NS and receive notifications).

FIGURE 2.3: Publish/Subscribe interactions in a NS

The basic model of interactions in a notification service is shown in Figure 2.4, where a NS is used to manage the subscriptions held by consumers, enabling the efficient delivery of notifications. The NS represents a service mediating between the publishers publishing notifications, and consumers consuming notifications. A consumer will register interest in notifications by subscribing, indicating the topic of the notifications they are interested in. consumers may not be aware of the sources of the notifications, only the topic to which they belong. Equally, publishers may not be aware of the number or identity of any consumers, as this information is managed by the NS. Consumers unsubscribe when they no longer wish to receive notifications.

There are two types of subscription in a NS — *push subscription* and *pull subscription*. In a push subscription, it is the job of the NS to deliver a notification to a consumer when it is published. Conversely, in a pull subscription it is the responsibility of the consumer to check for any new notifications on the NS. These subscriptions can be likened to

[1] http://www.microsoft.com/msmq/
[2] http://www-306.ibm.com/software/integration/wmq/

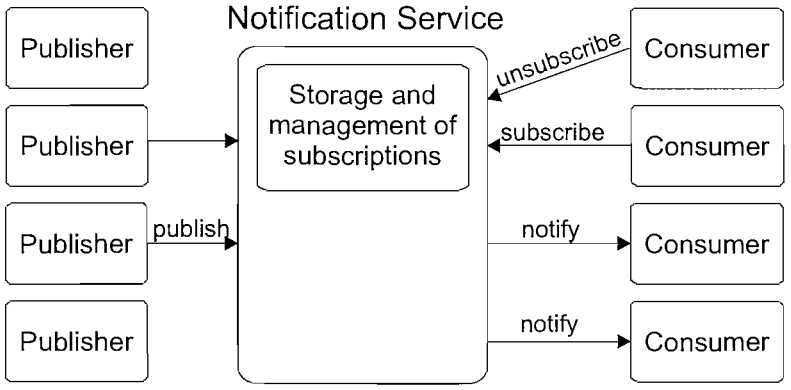

FIGURE 2.4: Interactions in a Notification Service

e-mail protocols — push subscriptions are like SMTP (Postel, 1982) where messages are delivered to waiting servers. Pull subscriptions likened to receiving your e-mail with a mail client using POP3 or IMAP, where the messages reside on a server while you are offline. Push subscriptions tend to be delivered as soon as a notification is published, hence are well-suited to time-critical applications. However, consumers may be located behind firewalls, with no way for a NS to contact them directly. In this situation, a pull subscription is the only way a consumer can receive any notifications.

The publish/subscribe interaction pattern decouples the publisher from the consumers in terms of time, space and synchronisation (Eugster et al., 2003). Time is decoupled as there is no need for the publishers and consumers to be active at the same time — notifications may be published while a consumer is off-line, in which case they will be delivered when the consumer reconnects later. In terms of space, there is no need for the publisher and consumer to ever contact each other directly, or even be aware of each other's existence. Notifications are delivered to the consumers by a third party, so publishers do not know how many consumers are receiving their notifications, or the identities of any of them. Equally, consumers do not necessarily know the source of the notifications that they receive. Decoupling of synchronisation is implied by the previous two properties — as the consumer and publisher do not contact each other directly, the processes of publishing and receiving notifications can be asynchronous non-blocking operations, allowing each to work on something else while not processing notifications. Decoupling the publishers and consumers in a system increases the scalability by reducing the explicit dependencies between the parties in a system, and enabling it to be more suited to a distributed environment, such as in a mobile environment (Huang and Garcia-Molina, 2001).

Notification services can provide additional functionality on top of the delivery of notifications between publishers and consumers, including these diverse possibilities:

- *Reliable message delivery* — As an increasing number of network-based applications rely on a MOM layer for communications, the need for guaranteed de-

livery of interactions becomes more important (Pallickara et al., 2005). This is shown in particular for the field of Web Services by the recent development of two standards, WS-Reliability (OASIS, 2004a) and WS-ReliableMessaging (Bilorusets et al., 2005). Reliable message delivery for a notification service ensures that a consumer subscribed to notifications on a particular topic is guaranteed to receive every notification published on that particular topic, and that failures in the middleware layer can be tolerated (Pallickara and Fox, 2004b).

- *Durable topics* — If a consumer connects to a NS and subscribes to a particular topic, they will typically receive all notifications sent on that topic from that point forward Krishna et al. (2003). Durable topics allow a consumer to subscribe to a topic and receive all existing notifications that were sent before the point of subscription.

- *Message prioritisation* — In some cases it may be important to prioritise certain messages, to ensure that important messages are delivered as soon as possible. NSs can enable notifications to be prioritised in such a way.

- *Message digests* — Consumers may not wish to receive a single notification for every event that occurs in a system; they may instead wish for a single digest of all notifications sent over a certain period to be created for them by a NS (Fox et al., 2005). Such a service can be extended further to provide arbitrary transformations of messages, such as translating a notification from one format to another, or combining notifications from multiple publishers to remove any duplicate notifications.

- *Message Filtering* — While a consumer registers an interest in notifications on a particular topic, they may not actually be interested in all of the messages on that topic. The NS may provide additional filtering, to restrict the notifications that the consumer receives to those that it is interested in (Banavar et al., 1999a; Carzaniga et al., 2000).

Notification services can be used to publish notifications of many different events, such as changes to a database (Oinn, 2002) or stock prices. Recently, they have also been used in projects from the grid community, including the following examples:

- myGrid (myGrid Project, 2003) is an e-Science project that aims to help biologists and bioinformaticians perform workflow-based *in silico* experiments, and automate the management of such workflows through personalisation, notification of change and publication of experiments (Moreau et al., 2003). The focus of myGrid is on increasingly data-intensive bioinformatics and the provision of a distributed environment that supports the in silico experimental process. This experimental

process is expressed as a workflow script, describing how services should be composed in order to realise the experiment desired by the scientist. As such workflows may take days or even weeks to complete, it is impractical to have a user agent on-line to monitor the status of their job. Instead, the *myGrid notification service* (MGNS) is used to forward messages to user agents when present, or to store and forward messages in their absence (Krishna et al., 2003).

- SERVOGrid (Donnellan et al., 2004; ServoGRID Project, 2005) is a Grid system for earthquake modelling and simulations. One application is RDAHMM (Granat, 2004), a time series data analysis program for mode change detection. In SERVOGrid there are a number of geographically distributed GPS stations, which publish data continuously. This data is used by the RDAHMM application, as well as other applications including a database for permanent storage, and portal applications for human interaction. Since data is published by a number of sources and consumed by a number of applications, this is ideal for a notification service. The NaradaBrokering notification service (Pallickara and Fox, 2003) is used to deliver notifications from the individual GPS stations to any applications that have expressed an interest in it. In turn, these applications may publish events based on the data from the GPS stations, which may be used through the notification service by other applications.

2.4.1 Quality of Service in Notification Services

Being able to request subscriptions to a particular set of notifications is sometimes not enough. As a notification service is generally asynchronous, there may be no explicit guarantees on how a service will be delivered. Service-oriented architectures are increasingly supporting Quality of Service (QoS) constraints to be set when a service is requested, and even when a service is discovered (Deora et al., 2004). Hence, a mechanism must be present to enable a subscription to be requested with a specified set of Quality of Service (QoS) guarantees. Such QoS terms specify *how* a service is delivered (*non-functional requirements*) rather than details about the subscription itself (functional requirements), and may be used to indicate properties of the subscription itself, or of services that a notification service may provide on top of the subscription (such as notification digests or aggregation). To illustrate the range of QoS terms, the following are all possible:

- *Notification Frequency* — The amount of time between notifications, for situations where the NS may store notifications to be delivered later.
- *Message Size* — Whether a message should be compressed, or restricted to a certain size before sending an additional message.

- *Notification Format* — The format a message should be in. NSs are capable of translating from one format to another.
- *Reliability* — Whether a guarantee is provided that a notification will be delivered.
- *Priority* — The priority a message is sent with. Higher priority messages may be delivered before lower priority messages ahead of them in a queue.
- *Timeliness* — The amount of time between an event occuring and a notification being sent.

Every notification service we have examined supports QoS to some degree by allowing subscriptions to be requested with certain conditions. However, consumers and publishers may not always agree on what QoS conditions a service should be provided under. Current NSs provide no mechanism for resolving this.

2.4.2 Distributed Notification Services

As the scale of a deployment increases, so too does the load placed on the NS. As with any single service in a system, if load increases too much it can become a bottleneck, potentially restricting performance or availability of the entire system (Pallickara and Fox, 2001). Hence, in large-scale deployments, multiple instances of a NS are likely to be hosted at different locations (Krishna et al., 2003), with such distribution of NSs offering many benefits. As there is no longer a single NS responsible for the delivery of all notifications, the system scales better to handle larger numbers of publishers, consumers and messages. It also tolerates one or more NSs failing without bringing down the entire system, thus increasing the scalability and reliability of the system. Additionally, this can increase security in a system by allowing information on private topics to be published only to a local NS situated behind a firewall, while being able to use the same NS for global topics.

Distributed NSs are networked, enabling them to propagate notifications between publishers and consumers that are connected to different NSs. It is the responsibility of the NS to handle the routing of messages between publishers and subscribed consumers. Figure 2.5 shows a scenario where there are many consumers subscribing to notifications being published by a single publisher. The consumers are distributed between many NSs, relying on their NS to deliver notifications. In a distributed NS, multiple publishers and consumers may be connected to different NSs, which in turn are connected to each other using any topology. A clear problem with a distributed NS such as this is how to optimise the number of notifications being sent between NSs (Pallickara and Fox, 2004a). For example, if a NS has ten consumers subscribing to notifications on the same topic, it is only necessary for the NS to receive a single copy of each notification. A *shared subscription* is a subscription made between notification services in

order to receive notifications that will be redistributed between multiple consumers with similar interests. The process of redistributing notifications received on this subscription is *sharing notifications*. For multiple consumers to be able to use a single shared subscription, the two subscriptions from the consumer must be compatible, i.e. the same topic, and compatible QoS conditions. For example, if one QoS constraint is message format, consumers requesting different formats will not be able to share a subscription.

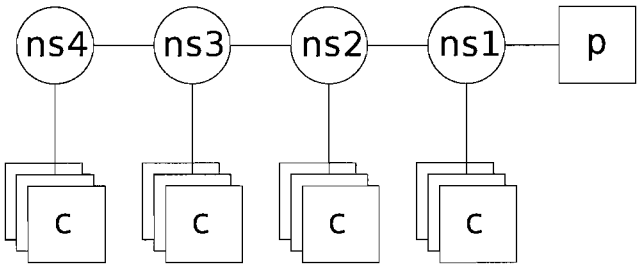

FIGURE 2.5: Consumers (c) connected through notification services (ns) to publishers (p)

There are many existing implementations of notification services, in both commercial and research environments. Many of these support deployment in a distributed configuration, as discussed in this section. In Section 2.6, we discuss the approaches taken by some existing NSs to the problem of sharing notifications.

2.5 Standardisation Efforts

With many different implementations of messaging-based products, it is difficult for them to work together. A number of efforts have been undertaken to enable different solutions to work together: WS-Notification is a standard being developed to enable web services-based notification services to work together; WS-Eventing is a similar but competing standard; JMS, while not a standard, is an API which many other implementations of messaging products support, providing some common ground on which they can cooperate. We consider each in detail below.

2.5.1 WS-Notification

WS-Notification (Graham et al., 2004) is a family of specifications making up part of the Web Services Resource Framework (WSRF) (Czajkowski et al., 2004) defining a web services approach to notification using a topic-based publish/subscribe model. It is made up of WS-BaseNotification (OASIS, 2005a) and WS-BrokeredNotification (OASIS, 2005b). WS-BaseNotification defines the basic roles necessary for publish/subscribe interactions — a *NotificationProducer* and *NotificationConsumer*.

A NotificationConsumer defines an endpoint for receiving notifications from a NotificationProducer. The WS-Notification standard defines two types of notifications that may be received — raw notifications and *notify* messages enriched with metadata. Raw notifications are intended to be used for an application-specific context, whereas notify messages provide additional information in a well-defined format, such as the topic of the notifications, and references to the subscription and producer.

A NotificationProducer produces notifications to be delivered to a NotificationConsumer on a particular topic. The NotificationProducer is also responsible for handling subscription requests and maintaining a record of which subscriptions are in place. The NotificationProducer handles NotificationConsumers subscribing to topics, but uses another entity, a *SubscriptionManager* to enable advanced subscription operations including unsubscribing, renewing, pausing and resuming a subscription.

WS-BaseNotification only defines the entities required for basic publish/subscribe notification, and makes no provision for a separate entity acting as a notification service — publishers and consumers must be directly connected instead. The specification of WS-BrokeredNotification enables support for distributed notification, primarily by defining a *NotificationBroker*, which is "an intermediary that allows publication of messages from entities that are not themselves service providers" (OASIS, 2005b). Essentially, it allows notifications to be sent from a NotificationProducer to a NotificationConsumer via any number of intermediaries, enabling advanced messaging features such as the publishing of notifications collected from multiple sources. As this specification is built to coexist with WS-BaseNotification, a NotificationBroker is both a NotificationProducer and a NotificationConsumer. Hence, as far as a NotificationConsumer is concerned, there is no difference between subscribing to notifications from a NotificationProducer and subscribing to notifications from a NotificationBroker. As WS-BrokeredNotification does not define what a NotificationBroker should do with a notification it receives, any number of services can be provided. A basic service would simply to forward notifications unaltered, but more advanced features are also possible, such as on-demand publishing, logging of notifications and transformation of notification topics and content.

Closely related to the WS-Notification family is *WS-Topics* (OASIS, 2004b), a standard for the hierarchical definition of topics for notifications. This allows NotificationConsumers to specify precisely which parts of a topic they are interested in. For example, a topic may be composed of several subtopics. If the NotificationConsumer is interested in enough of the subtopics individually, it can subscribe to the parent topic and receive notifications from all of the topics. Hierarchical topics also enable easier administration, for example administering security policies. WS-Topics is an XML format, defining topics as being unique within an XML namespace. A shared understanding of what each topic means is still required, such as an ontology (Gruber, 1993).

It is worth noting that at the time of writing, both WS-BaseNotification and WS-

BrokeredNotification have recently moved to the first public review stage, but WS-Topics is still in a draft form.

2.5.2 WS-Eventing

A similar but competing standard to WS-Notification is *WS-Eventing* (Box et al., 2004) produced by IBM, BEA Systems, Microsoft, Computer Associates, SUN Microsystems and TIBCO Software. Regarding the relationship between the competing protocols, the authors of WS-Eventing mention that WS-Eventing provides similar functionality to that of WS-BaseNotification.

In WS-Eventing, *subscribers* register their interest with a web service called an *event source* which provides *notifications*, although there is no formal specification of topics. Notifications are only in the form of raw, application-specific messages, rather than the metadata-enriched *notify* messages offered by WS-Notification. The subscription management interface of WS-Eventing is similar to WS-Notification described above, albeit with a lack of mechanism for defined topics. There is also no equivalent of a NotificationBroker.

WS-Eventing does contain features missing from an earlier version of WS-Notification such as support for pull notifications (Pallickara and Fox, 2005), but as both standards are still in draft or review states, it is still unclear which one will become more widely used. However, WS-Notification appears to be the more complete specification that is closer to becoming a full standard.

2.5.3 JMS

First released in August 1998, the Java Message Service (JMS) (Hapner et al., 2002) is an API enabling applications to send and receive messages, and to interoperate with other JMS-compatible messaging products. The original intention of JMS was to enable Java programs to interoperate with other MOM systems, such as IBM's Websphere MQ. *JMS Providers* are entities implementing JMS for a specific messaging product, and have been created for many popular MOM systems, enabling interaction between Java programs and a variety of MOM systems.

JMS supports both point-to-point messaging (sending a message from one component to another) and the publish-subscribe subscription model. A MessageProducer object is used to publish messages on a particular topic, which are then received by MessageConsumers. Messages can be published and received *asynchronously*. The JMS API also provides for *reliable* message delivery, ensuring that every message is received exactly once.

2.5.4 Summary

JMS is a mature API, with support in the form of JMS providers for many MOM systems available today. However, with the increasing movement towards web services, specifications like WS-Notification and WS-Eventing are important for the use of messaging platforms from web services. Of these specifications, WS-Notification is the more complete, but still lacks some of the more advanced MOM features supported by JMS, such as reliable messaging (Humphrey et al., 2004). However, both specifications still provide a substantial specification for using MOM systems from web services. Reliable messaging is being implemented by two further specifications from the same groups as WS-Notification and WS-Eventing — WS-Reliability (Web Services Reliable Messaging TC, 2004) from OASIS and WS-ReliableMessaging (Bilorusets et al., 2005) from IBM, Microsoft and other companies.

2.6 Existing Notification Services

There are many MOM platforms available today. Some of these are also notification services, of which there are also many examples, both from the commercial and research communities. We review existing notification services below.

2.6.1 myGrid Notification Service

The myGrid notification service (MGNS) is a messaging platform based on JMS, originally intended to provide asynchronous notifications between grid services for myGrid. However, as grid services have become more aligned with web services, MGNS can provide messaging capabilities to any web service. MGNS is built on top of an existing JMS server (OpenJMS), to which it delegates the basic messaging functionality, enabling MGNS to concentrate on providing additional features, including *durable topics* (where a consumer subscribing to a durable topic will receive all notifications ever published on that topic, including those from before it subscribed) and *virtual topics*.

MGNS supports deployment as a federated notification service, where multiple instances of MGNS each have a set of local topics (Krishna et al., 2004). These topics are then registered with a topic registry (Miles et al., 2005), marking them as a *member topic* of a *virtual topic*. Each MGNS instance participating in the virtual topic subscribes to each of the member topics. Consumers then subscribe to the local topic at their MGNS instance, and receive all message published on the virtual topic at any of the instances.

Notifications are shared in this system when a MGNS instance in a virtual topic subscribes to a member topic on another instance. Every time a notification is published, a single copy is sent between NS instances which is then shared between the consumers at

each instance. However, in this system, each MGNS instance in a virtual topic subscribes to every other instance, which could cause a scalability problem in a large deployment. However, the service supports subscriptions through intermediate instances, for example to facilitate connection through a firewall.

2.6.2 NaradaBrokering

NaradaBrokering[3] (Pallickara and Fox, 2003) is a high-performance distributed brokering system, which provides support for centralised, distributed and peer-to-peer interactions. It is a JMS compatible system (Fox and Pallickara, 2002) supporting audio and video conferencing (Uyar et al., 2003), integrated performance monitoring and communication through firewalls. The main focus of the work on NaradaBrokering is efficiently handling the issues of scaling, load balancing and resilience.

NaradaBrokering uses *brokers*, which are organised into clusters, in turn organised hierachically. The brokers have a *Broker Network Map* (BNM), making each broker aware of the broker network layout enabling efficient routing of messages to different destinations. As the system is intended to support a large number of brokers, each broker does not need to know the topology of every other broker in the network, only an abstract view of the network that still enables them to calculate optimal paths. Changes to the broker network (brokers joining, leaving or moving) are only propagated to those brokers whose BNM would be affected. The process of computing destinations for each message is referred to as the *matching* of events, and a significant amount of work has gone into creating efficient matching algorithms for the system (Pallickara and Fox, 2004a). Notifications being passed between brokers can be shared between multiple subscriptions, since the matching algorithm determines consumer interest in a topic, and the notifications are also shared between the consumers by the matching algorithms. However, NaradaBrokering enables its consumers and publishers to connect to any broker in the network, making it difficult to explicitly share subscriptions over a predefined route through the broker map.

2.6.3 Other Notification Services

Elvin (Segall and Arnold, 1997) is a notification service developed at the University of Queensland. It supports federation of Elvin servers to enable the system to scale beyond the limits of a single server, with each server subscribing to the other servers in the federation. To reduce the amount of redundant data being transmitted, *quench* messages are sent when there are no subscriptions at a particular server. These messages contain updates to a list of filters which will catch notifications a server is *not* interested in (Segall et al., 2000).

[3] http://www.naradabrokering.org/

Gryphon[4] from IBM is a broker network which was designed to enable the efficient distribution of notifications in a content-based subscription system, where arbitrary filters can be applied to content, as opposed to subject-based subscriptions where all data on a particular subject or topic is delivered. The problems identified with such a system are how to match an event to a large number of consumers, and how to multicast the events to the consumers within the network (Banavar et al., 1999a). In Gryphon, brokers use *link matching* to maintain partial lists of subscriptions at each broker. When a message is published, each broker partially matches events against the subscription list at each hop in the network. Shared subscriptions are used to only send a single message over each hop of the network, providing an efficient method of disseminating the message between multiple consumers.

Siena[5] is another distributed notification service based on the principles of IP multicast, and its objectives are to route notifications in one copy as far as possible, as close to the client as possible. As the consumers specify filters on the information they require, this information is replicated as close to the sources of notifications as possible, so that subscriptions can be shared (Carzaniga et al., 2000).

The Object Management Group (OMG) has produced the CORBA Event Service (Object Management Group, 2004), which supports channels where a producer can create an *event* on that channel that will be received by any consumers listening on that channel. This was extended by the creation of the CORBA Notification Service (Object Management Group, 2002), adding functionality in order to allow applications to send messages to objects in other applications without any knowledge of the receiving object's existence. The notification service uses an event channel so that a *supplier* can publish messages to any number of *consumers*, without knowledge of the consumers, or even whether there are any consumers, and provides event filtering and QoS capabilities. At the application level, the consumer of one channel can be a publisher for another channel, enabling an implementation of a distributed notification service similar to that used in MGNS. However, there is no explicit system-level support for the sharing of subscriptions.

Commercial message queuing products such as IBM's Websphere MQ and Microsoft's MSMQ are primarily aimed at MQM applications rather than a subscription-based approach to delivering notifications to multiple consumers. However, publish/subscribe interactions are supported in Websphere MQ (Perry et al., 2001), which facilitates publish/subscribe operations in a distributed manner, enabling publishers and consumers to use different brokers and have notifications routed between them. Websphere MQ also supports persistent and reliable delivery of notifications.

[4]http://www.research.ibm.com/distributedmessaging/gryphon.html
[5]http://serl.cs.colorado.edu/~carzanig/siena/

2.7 Summary

In this chapter, we have pointed out that many current distributed systems, commonly based on web services technologies, can make use of a Message-Oriented Middleware platform in order to facilitate asynchronous communications between entities in such a system. In systems where notifications about a particular subject need to be distributed to many consumers, a notification service using the publish/subscribe model of interactions is a suitable delivery mechanism. Notification Services can be used for publishing any event-based information, such as changes to the contents of a database, or notification that a real-world event has occurred.

As the scale of a NS deployment is increased, distributed NSs become useful in order to solve such problems as load balancing, reliability and security. A distributed NS is a network of NSs interlinked so that consumers and consumers may be distributed throughout the network, publishing notifications to, and consuming notifications from, different instances of a NS. A distributed NS handles the routing of messages between publishers and consumers. When multiple consumers are subscribed to the same topic at the same NS instance, that instance may make a single shared subscription to receive the data from the publisher, sharing the notifications between its consumers.

2.8 Discussion

Notification Services enable consumers to request a desired Quality of Service from the NS, which can guarantee certain conditions about the delivery of notifications. However, if all consumers are allowed to request any levels of QoS, it may be possible for a small number of consumers to overwhelm a service by requesting significantly high QoS conditions. As differences may exist between the preferences of a consumer and a publisher over QoS levels, a mechanism of finding a compromise between the two different preferences is required. Negotiation of QoS conditions is already used in other fields such as distributed real-time systems to specify requirements for resources (Li and Ravindran, 2004), and in multimedia systems for resource reservation (Rothermel et al., 1997). Hence we see negotiation as a suitable mechanism for resolving differences in the preferences of consumers and publishers, but no existing NSs support negotiation for this purpose.

In a distributed notification service, when many consumers at a particular notification service subscribe to notifications on the same topic with similar levels of QoS, the notification service can share the subscription between multiple consumers. However, if these levels of QoS are significantly different, it may be impossible to share the subscriptions. In this case, extra copies of notifications may be unnecessarily transmitted between the notification services. Negotiation could be used here to persuade additional consumers

to request QoS levels compatible with the existing subscriptions that are held.

In the next chapter we will discuss negotiation, and review different negotiation models to find one suitable for use in this context of a notification service. A suitable model for negotiation would enable the developers of a notification service to support negotiation over QoS levels *without* needing to be knowledgeable in negotiation techniques, and without causing significant modification to the interaction patterns used by the notification service.

Chapter 3

Automated Negotiation

In a notification service where consumers are allowed to request levels of quality of service (QoS) when they subscribe to a particular topic, there can often be a difference between the levels of QoS a service provider is willing to provide, and the levels the consumer would like to receive. Automated negotiation can be a solution to resolving these differences and producing a compromise between the two, enabling the consumer to still receive an acceptable level of QoS while the service provider is not overloaded by providing services of a quality that is difficult to maintain for many consumers. In this chapter we review automated negotiation.

3.1 Introduction

Recent trends in computing have seen computer systems consisting of many different components distributed across networks (such as grid systems (Foster and Tuecke, 2001)) and agent-based systems (Jennings, 2000)) and working together to accomplish their goals. As much of the work on automated negotiation comes from the field of agent-based computing, we choose to adopt a view of agent-based systems for the purpose of explanation, in which an agent has the following characterisation (Wooldridge, 1997):

> "an agent is an encapsulated computer system that is situated in some environment and that is capable of flexible, autonomous action in that environment in order to meet its design objectives"

An agent's goal may be comprised of subgoals or tasks that it is not capable of achieving itself, instead requiring the use of a service provided by another agent. Alternatively it may be capable of achieving the subgoals itself, but less efficiently or at a higher cost than by using another agent. To enable the goals to be achieved efficiently, the agents need to be able to *interact* in order to *co-ordinate* achievement of the shared task.

Negotiation is the process by which two or more parties communicate in order to reach a mutually acceptable agreement on a particular matter (Jennings et al., 2000). It can be described as a joint search over a problem space with the goal of reaching a consensus (Guttman and Maes, 1998). Negotiation is key to managing the interactions between autonomous agents because, by definition, autonomous agents will do what they want to do. For one agent to obtain a service from another, it must first convince the agent providing the service to cooperate on the task. In its simplest form, negotiation involves asking for a service, and receiving a positive or negative response. Negotiation may also include iterating sequences of offers and counter-offers, and rewards or payment for the item under negotiation. After a successful negotiation, a mutually acceptable set of conditions for the supply of an item or service will have been reached.

In the context of a notification service, where participants can request a subscription to notifications on a particular topic with constraints specified on various QoS issues, the negotiation item represents the subscription to notifications on a specific topic. The issues under negotiation represent various QoS conditions, such as the frequency of notifications, the granularity of the information contained in the notifications, the length of the subscription and the price paid for the subscription. In general, a publisher and a consumer will tend to have conflicting preferences over many of these conditions. For example, a consumer may wish to have frequent notifications of an event, and may wish to receive these notifications without paying too much for them. However, a busy publisher may wish to limit the frequency that notifications are sent out in order to reduce its load, and may additionally like to receive a payment for the services it provides. Hence, negotiation can be used to find a mutually acceptable set of conditions under which a subscription to notifications may be delivered.

In this chapter we review existing work on automated negotiation. To give this context, we first discuss different mechanisms for negotiation between humans or agents in Section 3.2. In Section 3.3 we discuss the different approaches taken to automated negotiation. For the use of automated negotiation to become more widespread, standards for negotiating and the formation of agreements are required, as discussed in Section 3.4. Additional related work is discussed in Section 3.5, before summarising in Section 3.6 and discussing how this relates to our aims in Section 3.7.

3.2 Negotiation Mechanisms

Negotiations occur for one party to obtain a *negotiation item*, which could be an object or a service, from another. While trying to obtain this item, various *negotiation issues* are proposed — attributes about the item or the constraints on its delivery, over which participants usually have conflicting *preferences*. Preferences define the desired outcome of the negotiation for each participant.

There are many *mechanisms* by which negotiation can take place, which can be equally applied to autonomous agents as they can to human interactions, though we use the term agent in the descriptions of each. In order to summarise the different types of negotiation, a distinction can be drawn based on the number of participants in a negotiation. Three different groups of negotiation techniques are produced using this classification: one-to-one, one-to-many and many-to-many techniques.

3.2.1 One-to-one Negotiations

In a one-to-one negotiation, a single agent attempts to obtain an item from another single agent. Both agents may have conflicting preferences over the conditions under which they would like the transaction to take place, so a mutually acceptable set of conditions must be found before a deal can be reached. *Bilateral* negotiation (negotiation between two agents) mainly takes the form of *bargaining*: a situation in which individuals have the possibility of concluding a mutually beneficial agreement, there is a conflict of interests over which agreement to conclude, and no agreement may be imposed on any individual without their approval (Osborne and Rubinstein, 1990).

Essentially, bargaining consists of making alternating offers, or *proposals*, (i.e. potential agreements) containing values for each of the issues in a negotiation (Larson and Sandholm, 2002). Behaviour of a participant upon receipt of a proposal varies with different negotiation mechanisms, but has the basic requirement of accepting or rejecting the proposal. Rather than rejecting a proposal outright, a common behaviour is to offer a *counter-proposal* in return, allowing concessions to be made by both parties in a negotiation, hopefully speeding up the process of finding a mutually acceptable condition under which an agreement may be formed. Bargaining enables multi-attribute negotiation (such as over price, delivery time, etc.), rather than negotiation over a single attribute, a typical property of auctions (described in Section 3.2.2).

An extension to the model of bargaining is argumentation-based negotiation (ABN) (Rahwan et al., 2004; Kraus et al., 1998). In ABN, participants can exchange additional information with a proposal in order to argue, justifying their negotiation stance, or attempting to persuade their opponent to change his negotiation stance. Jennings et al. (1998) use the analogy of negotiation as a search through a multi-dimensional space of potential solutions, where the number of issues in the negotiation controls the number of dimensions of the space. Negotiation is thus a distributed search through this space of solutions, attempting to find a solution considered by all participants to be acceptable. The minimum capabilities needed to negotiate are the abilities to propose some part of the agreement space as being acceptable, and the ability to respond to such a proposal indicating whether it is acceptable or not. A proposal (as defined above) marks a single point in the solution space, rather than a region, and if agents can only accept or reject these proposals, negotiation is very time consuming. However, if counter-proposals are

offered, the recipient of a counter-proposal can attempt to infer the preferences of its opponent from the change between the proposal and counter-proposal. In ABN, meta-data sent with a proposal rejection or a counter-proposal offers a critique of *why* the proposal was not acceptable, indicating to the recipient of the critique which regions of the agreement space *are* acceptable.

While an opponent may try to influence his opponent's negotiation stance by suggesting that they change the values they consider acceptable, ABN also allows dynamic modification of the set of issues under negotiation — for example, when negotiating over the purchase of a new car, the dealer may choose to fit a car alarm free of charge as an incentive to make the customer agree on a price. This may not have been an issue earlier in the negotiation, but the inclusion of the additional issue can make forming an agreement easier. ABN represents a significant ongoing research effort (Sierra et al., 1997b; Parsons and McBurney, 2003; Amgoud and Maudet, 2002; Karunatillake and Jennings, 2004).

3.2.2 One-to-many Negotiations

In a one-to-many negotiation, a single agent attempts either to obtain or make available a service or product to one of a number of agents. The most common type of one-to-many negotiation is an *auction*, defined as a bidding mechanism, described by a set of auction rules that specify how the winner is determined and how much he has to pay (Wolfstetter, 1994). Auctions have become commonplace on the Internet, with sites such as eBay[1] being used for thousands of items daily, and provide a mechanism of selling an item to a bidder who values it the most (i.e. who bids the most). There are several different types of auction, but a big distinction lies between single-sided and double-sided auctions. Single-sided auctions are those in which a single seller accepts bids from multiple buyers, whereas in double-sided auctions, multiple buyers and multiple sellers submit bids at the same time. Double-sided auctions are actually many-to-many negotiations and are discussed in Section 3.2.3. Although many different types of single-sided auctions have been created, there are four main types (Klemperer, 1999), described below:

- English Auction (Ascending bid)

  In an English auction, a starting bid is offered by the auctioneer. Buyers then place a bid on the item. An unlimited number of bids can be made, with the restriction that each new bid must exceed the current high bid by a specified amount. When no more bids are received, the item under auction is sold to the current high bidder at the final price.

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

- Dutch Auction (Descending bid)

  Dutch auctions are the opposite of English auctions: an auctioneer starts with an initial high offer, and calls out decreasing offers. The first bidder to accept the offer made by the auctioneer wins the item.

- First price, Sealed bid

  In a first price, sealed bid auction, each bidder submits a single bid privately, so the other bidders do not see it. The auctioneer then sells the item to the highest bidder.

- Vickrey auction (Second price, Sealed bid)

  A Vickrey auction is similar to first price, sealed bid, with the difference that the item is sold to the highest bidder, but for the price bid by the second-highest bidder. This minimises the effect of *winner's curse*, where the winner in an auction often ends up overpaying for the item they have won. The Vickrey auction encourages bidders to bid their true estimation of the item's value.

Variations on English auctions are often used for an *online auctions* (Anthony, 2003), typically to increase the flexibility of the auction. In an auction, participants gather in one room, and the auction lasts a few minutes, requiring quick decisions. Online auctions can run for days, with people able to participate remotely.

3.2.3 Many-to-many Negotiations

In many-to-many negotiations, several buyers and sellers submit bids for an item simultaneously. An example of this type of negotiation is a Continuous Double Auction (CDA) (Friedman and Rust, 1993). In a CDA, many buyers and sellers continuously submit bids to buy and sell items, and bids are matched between buyers and sellers continuously throughout the period of the auction. Trades occur without terminating the negotiation, as opposed to auctions in which the formation of an agreement terminates the negotiation. CDAs are widely used in the trading of stocks and commodities, such as shares in the New York stock exchange (Friedman, 1993).

3.2.4 Summary

Many different mechanisms for negotiation have been proposed, and we have presented a summary of some of these mechanisms above. The context we are examining is a notification service, where consumers subscribe to notifications on a particular topic while negotiating over QoS constraints. While in a distributed system there may be many potential service providers which could be used, often in scientific fields such as

bioinformatics, scientists may have varying levels of trust in different service providers, and therefore dictate themselves which one should be used. Hence, we consider the process of selecting partners to be separate from that of negotiating with them, which is a one-to-one interaction between a consumer and a publisher. Each party holds conflicting preferences over the constraints associated with the subscription. Bargaining and ABN both allow a consumer and a publisher to agree on a set of mutually acceptable constraints over which a subscription will be formed. Bargaining enables the two parties to exchange proposals in order to find this set of constraints, and ABN also allows arguments to be used to strengthen or change a party's stance within the negotiation.

3.3 Approaches to Automated Negotiation

As described in the previous section, there are many different types of negotiation. For two or more *autonomous agents* to use negotiation to cooperate on some activity, *automated negotiation* is required. Research in automated negotiation is concerned with enabling autonomous agents or entities to negotiate with each other with no human input. Automated negotiation can be split into two main topics: negotiation protocols and strategies.

Negotiation protocols describe the set of rules governing a particular type of interaction (Rosenschein and Zlotkin, 1994; Jennings et al., 2001). In an automated negotiation, a negotiation protocol defines the negotiation mechanism that will be used, and how a participant can act in the negotiation. This covers the types of participants allowed, the valid negotiation states, and the actions that may be taken by the participants in each state, such as which messages can be sent by each party. Specifying a negotiation protocol enables a participant to act within the rules of a negotiation, but does not specify how it should behave within these conditions.

A *negotiation strategy* defines how a participant may act in a negotiation, within the rules specified by the protocol. While any action that complies with the protocol is permitted, there are many possible courses of action that would lead to a poor result. For example, it is normally assumed that an agent participating in a negotiation is rational — it will always act to try and increase the benefit to itself, or to the system as a whole. It is therefore irrational for an agent trying to purchase an item for the best price to propose a price significantly over its valuation of the item as an opening offer. However, it is still valid to do so according to the negotiation protocol. The negotiation strategy of an agent is referred to as its decision-making model in Jennings et al. (2001), and is defined as the decision making apparatus the participants employ in order to act in line with the negotiation protocol in order to achieve their objectives. We adopt this definition for the rest of this work.

It should be noted that different approaches to automated negotiation put varying de-

grees of emphasis on both the protocol and the strategy. As an example, consider a Vickrey auction (discussed in Section 3.2.2). Vickrey auctions have a *dominant strategy* (a strategy that always yields a better payoff than any other strategies, regardless of the behaviour of opponents): an agent should always bid what it considers the item to be worth. Following this strategy will always lead to the optimal outcome, hence there is no reason for a rational agent to deviate from it. Under this assumption, it may be possible to specify this behaviour as part of a negotiation protocol. In other situations, the negotiation strategy is considerably more important. For example, when an agent bids in multiple auctions of different types simultaneously, a good negotiation strategy is required to maximise the chances of acquiring all of the desired items (He and Jennings, 2004; Sierra et al., 1997a).

In addition to the above, automated negotiation requires an understanding of various concepts to be shared between the participants in the negotiation. In particular, every participant should have a shared understanding of what the *negotiation item* (the object or service they are negotiating over) is, and the different *issues* (attributes of the item that are under negotiation). For example, when negotiating with a notification service for the provision of notifications, a subscription to notifications on a requested topic is the item under negotiation, and the issues represent the QoS aspects of the subscription being negotiated over, such as the frequency of notifications, granularity of information or the price paid for the subscription. All participants should have the same definition of frequency of notifications and the requested topic, otherwise the client may obtain something that it was not expecting. An *ontology* is a specification of a concept (Gruber, 1993), and enables the participants to share a common definition of the concepts involved.

3.3.1 Game-theoretic Approaches

Game theory (Osborne and Rubinstein, 1994) is a branch of economics concerned with interactions between self-interested agents, based on work described in von Neumann and Morgenstern (1953). It was originally aimed at the interactions between self-interested people, but is equally applicable to interactions between autonomous agents, as rational agents try to maximise the expected utility of any outcome (Simon, 1955), making them self-interested.

Essentially, game theory is concerned with an agent selecting the best or most rational strategy out of all possible strategies, taking many factors into account including, but not limited to, the behaviour of other agents in the negotiation, its own preferences and its estimate of private valuations held by its opponents. However, the space of possible strategies is extremely large, and searching this space is often computationally intractable. Classical game-theoretic approaches have been based on the assumption of unbounded computational resources being used to search this large problem space (Dash

et al., 2003), but this assumption rarely holds, as negotiations must often be completed before a deadline, or may be performed by a device with limited computational power.

Game-theoretic models have some disadvantages making them hard to apply to some fields. They rely on the assumption that it is possible to characterise an agent's preferences over a set of possible outcomes, which is often a difficult task, especially with multiple negotiation issues, as an agent's preferences may be based on private values, or may change over time as a negotiation progresses (Jennings et al., 2001). As such, game-theoretic models are more suited to negotiations where preferences are obvious, rather than more complex multi-attribute negotiation. Game theory tends to produce highly specialised models tailored to a specific negotiation mechanism — it is difficult to generalise the decision-making process. Such models often assume that all participants in a negotiation are completely rational, fully informed and have access to large amounts of computational power. Unfortunately, this is rarely the case — there are often restrictions on the resources available, and agents are not fully aware of their environment and opponents.

3.3.2 Heuristic-based Approaches

Game-theoretic approaches to automated negotiation aim to produce optimal solutions by searching the entire space of solutions. As this is both computationally hard and sometimes impossible due to incomplete awareness of the participants, an alternative is a *heuristic-based* approach. With such an approach, the aim is to find a good solution rather than the optimal solution. The methods of generating these solutions may be approximations based on game-theoretic approaches, or on more informal negotiation models (e.g. Raiffa (1982)). Heuristics allow the assumptions about access to resources and knowledge of the domain to be relaxed (Rahwan et al., 2004). For example, it is not always possible to fully model an opponent, or to consider the full space of possible outcomes. Heuristic-based approaches are based on realistic assumptions, rather than by searching for a fully-optimal solution. This makes them suitable for a wider range of domains, where the negotiating agent does not need to know the intricacies of the domain in which it is negotiating. This also makes it possible to use a heuristic-based approach to develop a reusable negotiation component which can be used in different domains without needing to be given additional information about its domain. Hence a heuristic-based approach is an ideal solution to automated negotiation for a system whose domain has not been fully defined.

However, heuristic-based approaches do suffer from some disadvantages. Due to their simplified model in comparison with a game-theoretic approach, they do not examine the entire space of potential solutions and may find an outcome that is sub-optimal (Jennings et al., 2001). Additionally, it is difficult to predict how such systems will behave, so they need extensive evaluation in order to determine whether the outcome will be satisfactory.

Two examples of heuristic-based approaches to automated negotiation are given below.

Faratin (2000) describes a bilateral negotiation model called Negotiation Decision Functions (NDF) using a heuristic-based approach. NDF clearly defines a negotiation protocol, specifying the rules of the negotiations. The strategies used are separate, allowing this to be varied independently of the protocol, and are based on combinations of functions called *tactics* that generate a value for one issue in the negotiation based on a single criteria. For example, time-dependent tactics use the amount of time remaining before the deadline to control their rate of concession. Resource-dependent tactics use the amount of a particular resource remaining to control their concession. Using these functions it is possible to create a negotiation strategy that can work in multiple domains without knowledge of the domain. Domain-specific information such as resource levels can be supplied by an external party, but negotiations are possible without such knowledge. Proposal evaluation is handled by *utility functions*, returning a value for an issue based on an agent's preferences. Utility is a measure of how good an agent considers a particular outcome. Utility functions control an agent's valuation of proposals, and can be implemented as simple linear functions or complex functions taking multiple factors into account.

Barbuceanu and Lo (2000) describe another example of a heuristic-based approach to automated negotiation, which finds the pareto-optimal solution for a negotiation (a solution for which it is impossible for one party to increase their utility without a corresponding decrease in their opponent's utility). This is done by generating every possible solution using all possible values for each issue, then evaluating each proposal and ranking them in order of utility. The best solution is then proposed to the opponent. If this is rejected, the opponent repeats the process, trimming the set of potential solutions after taking into account the received proposal. Using this approach it is always possible to find a solution that is optimal for both agents. However, this is an exhaustive process and, for multiple issues, requires a very large number of proposals to be exchanged. It is a computationally- and time-intensive solution. Optimisations have been suggested for this approach, including using probabilistic modelling to identify proposals likely to be accepted, although this comes with the risk of sacrificing the ability to find the pareto-optimal solution.

3.3.3 Argumentation-based Negotiation

Argumentation-based negotiation was introduced earlier in Section 3.2.1 as a negotiation mechanism for an agent to provide feedback justifying its negotiation stance or to attempt to influence the stance of its opponent. To enable automated ABN, negotiation protocols are required allowing this meta-information to be passed alongside proposals and counter-proposals. Sierra et al. (1997a) describe the augmentation of an existing negotiation protocol with the ability to pass this information, while McBurney and Parsons

(2004) describe an argumentation protocol to be used with other protocols. Jennings et al. (2001) makes the point that the transmission of this information can be seen as moving from a negotiation protocol into an argumentation protocol (e.g. Amgoud et al. (2000)), and then back again to the negotiation protocol when the argumentation dialogue terminates.

The reasoning of agents employing ABN techniques can include models based on beliefs, desires and intentions, such as shown in Parsons et al. (1998). A detailed review of ABN frameworks and systems is given in Rahwan et al. (2004).

3.4 Standardisation Efforts and Automated Negotiation

Although there have been many different efforts to produce automated negotiation frameworks (Bartolini et al., 2005; Faratin, 2000), these frameworks are typically separate from and unrelated to each other and unable to interoperate with each other. To facilitate the wider uptake of automated negotiation techniques, standards are required to define the protocols and interactions with which to negotiate. Any new work aimed at a field such as web services should be aware of relevant standards. Two major standards in this area are WS-Agreement and FIPA, which we describe below.

3.4.1 WS-Agreement

WS-Agreement (Andrieux et al., 2004) is the Global Grid Forum's (GGF) standardisation effort, comprising an XML-based protocol for the representation of agreements, an interaction protocol for establishing these agreements, and an interface for monitoring agreements already in place. WS-Agreement has arisen due to the frequent requirement in distributed service-oriented environments for a consumer to be able to request a service with a guaranteed QoS from a service provider, and for a mechanism to monitor compliance with guaranteed QoS levels to be available.

In WS-Agreement, an agreement between a service consumer and a service provider specifies one or more service-level objectives as expressions of requirements on the part of the consumer, and as an assurance about the availability of resources on the part of the service provider. Agreements include information about the service definition, as service objectives are often related to the definition of the service. Agreements are made up of the following information: service definition terms; guarantee terms specifying service level objectives; and an agreement context comprising information about the agreement parties and any relevant prior agreements.

WS-Agreement has been aimed at forming agreements in service-oriented distributed systems, such as job submission of computing jobs, or establishment of a set of QoS

terms for access to a particular service. As such, the protocol also supports the monitoring of agreement compliance. However, WS-Agreement makes no assumptions about the methods by which agreements are formed — it is independent of any negotiation mechanism.

Although WS-Agreement is still in draft, Ludwig et al. (2004) present an implementation of a WS-Agreement based architecture for creating and monitoring agreements for a service-oriented system, specifying the interface for domain-specific components such as system monitors.

3.4.2 FIPA

As negotiation is a form of interaction between involved agents, any negotiation framework requires a language with which to communicate. As most automated negotiation research is agent-based, an Agent Communication Language (ACL) is a suitable basis for a negotiation interaction protocol. The two major ACLs are FIPA ACL and KQML (Labrou et al., 1999).

The Foundation for Intelligent Physical Agents[2] (FIPA) is an organisation aimed at producing standards for the interoperation of heterogenous software agents. At the time of writing, FIPA had defined over 90 specifications. Of these, there are many interaction protocols which can be used to enable negotiation between autonomous agents, specified in the FIPA Agent Communication Language (FIPA, 1998).

The FIPA specifications are limited to the interaction protocols — they do not impose any restrictions on the strategies employed within these interactions. For example, in an English auction, the interaction protocol specifies that a bidder in an auction should propose an amount, and the auctioneer will either accept or reject that proposal. It imposes no requirement that the bid must be higher than the current highest bid, one of the main rules of an English auction. Thus, any system designed to use one of the FIPA negotiation protocols is designed to use a predetermined negotiation mechanism. FIPA interaction protocols are defined for many different types of negotiation including, but not limited to, English auctions, Dutch auctions and proposals in bilateral negotiations.

3.5 Other Related Work

Bartolini et al. (2005) highlight that current efforts for the standardisation of negotiation impose restrictions on the types of negotiation supported, or the knowledge of a negotiation protocol that is required. They propose an approach requiring all aspects of a negotiation mechanism to be formally specified and explicit, and focus on the full,

[2] http://www.fipa.org/

formal specification of negotiation protocols. Whereas some negotiation protocols such as WS-Agreement (Andrieux et al., 2004) specify the interaction protocol between parties involved in a negotiation, Bartolini et. al. stresses the need for additional protocol information, for example constraints on the amount by which a bid must increase, to be included in the specification. Without this information, the choice of a negotiation mechanism becomes implicit when a system is designed. By formally specifying all of the rules about a negotiation mechanism, it becomes possible to design a system that can support multiple negotiation mechanisms, and not be tied to any one in particular, potentially enabling trade and interactions with a wider range of consumers or service providers. A key part of creating the software framework for automated negotiation is the presentation of an abstract view of a negotiation process, taken from an analysis of many different negotiation mechanisms, both automated and human. In this abstract view of negotiation, a negotiation host facilitates the negotiation by providing communication mechanisms, using the analogy of a blackboard with controls over write access and visibility. Participants in a negotiation require admission, involving checking of credentials and the presentation of the rules for that negotiation. A *negotiation template* is shared between the participants, describing the conditions of the negotiation. During the course of the negotiation, the participants exchange proposals, expressing constraints over some or all of the conditions specified in the negotiation template. These proposals are sent to the negotiation host, which checks that they comply with the rules of the negotiation. Agreement formation rules are used to convert proposals into agreements when certain conditions are satisfied, and termination rules describe the conditions which will cause a negotiation to end. This abstract process is then extended to give a taxonomy of rules for negotiation, specified using the FIPA ACL (FIPA, 1998), which form part of an implementation of a software framework.

Cremona (Ludwig et al., 2004) is an architecture for the creation and monitoring of agreements based on WS-Agreement. Although this is more concerned with the establishment and monitoring of agreements than arriving at an agreement using negotiation, the application here is similar to our intention of negotiating over QoS conditions prior to the subscription to a notification service. RFC 1782 (Malkin and Harkin, 1995) describes a simple extension to the Trivial File Transfer Protocol (TFTP) in which options can be negotiated prior to a file transfer.

Tu et al. (1998) describe a way of dynamically adding negotiation abilities to an agent at run time using a pluggable architecture, sharing the negotiation capabilities between communication modules, protocol modules and strategy modules. This enables a program to load components enabling negotiation at runtime, determining the need for them dynamically.

3.6 Summary

Negotiation among computer programs and agents is an established field, and as such is supported by a wealth of literature. There are already different mechanisms for negotiation, whether one-to-one, one-to-many or many-to-many.

With the increasing popularity of distributed computer systems, such as those using the Grid and multi-agent systems, automated negotiation has become a real need to facilitate the cooperation and interactions of two or more autonomous systems, and to find a compromise when two parties have conflicting beliefs about the constraints of a service delivery. In this chapter, we have presented different approaches taken to tackling the area of automated negotiation.

3.7 Discussion

In a notification service where a consumer can request a particular level of QoS, automated negotiation would enable differences between the preferences about the level of QoS to be provided between the consumer and the service provider to be resolved. For use in a distributed system, any negotiation mechanism has to be able to work within computational and time constraints, in order that the addition of negotiation abilities does not prove detrimental to the primary function of the system (providing notifications in the context of a notification service).

Game-theoretic approaches typically assume access to large amounts of computational power, and that it is possible to characterise the preferences of an agent and its opponent. They also assume prior knowledge of the domain in which they are negotiating, and while it is possible to define some potential QoS conditions that can be negotiated, this would be insufficient for an architecture enabling service providers to offer QoS conditions that are unknown to the negotiation model. Hence, game-theoretic approaches are unsuitable for QoS negotiation in a distributed system.

Heuristic-based approaches can be used to negotiate in a domain of which the negotiating agent has no prior knowledge. It also imposes a low computational burden. Argumentation-based approaches tend to use strategies based on prior knowledge of the domain, although they can negotiate more efficiently than heuristic-based approaches due to being able to justify their stance to their opponent. Both approaches are valid for use with our proposed architecture, but as heuristic-based approaches more suited to unknown domains, they are the most appropriate choice. Due to similarities in the protocols used for ABN and heuristic-based approaches, we also argue that a design using a heuristic approach would not preclude the use of ABN in the future.

A generic automated negotiation framework was discussed in Bartolini et al. (2005) in

which many different negotiation mechanisms could be supported, but the implementation is based on a centralised *negotiation host*, which controls the negotiation process. One of the goals of a distributed notification service is to improve the scalability and reliability of a system, and relying on a central host for negotiation would contradict this aim. However, we recognise that this approach has the advantage that it can support additional negotiation protocols that were not considered at design time. The aim of negotiating over QoS in a notification service is to reduce load placed on a service provider while enabling clients to receive a high QoS. Thus, any negotiation solution should not place too much load on the service provider, otherwise it would negate the benefit of using negotiation in the first place. Although the model used in Barbuceanu and Lo (2000) always finds a solution that is optimal for both participants, it assumes an unlimited amount of computational power and time to do so. For this reason, it is unsuitable for such a use. The model in Faratin (2000) uses predefined negotiation strategies, some of which need no knowledge of the domain they are operating in. Hence, negotiations can be constrained to finishing within a certain time, which is a useful property when trying to set up a subscription for a time-critical task in a notification service. Using this model, the notification service or service provider would be able to influence negotiation process without getting involved in the details of negotiation, for example by supplying current resource levels. This model is thus well suited to use as the basis for enabling negotiation over QoS in a notification service, and we will further develop this in Chapter 4.

The current standardisation efforts of the Global Grid Forum in the form of the WS-Agreement standard are an important move towards enabling negotiation between a wider range of different systems. It is important that any model of negotiation being designed for use with web services or grid systems be aware of any relevant standards in the area. As WS-Agreement does not specify anything about the process of making agreements, it is not relevant to the actual process of negotiation. However, it could be used as the format for exchanging agreements and proposals. The concepts in NDF, which was created before the existence of WS-Agreement, are compatible with the concepts in WS-Agreement. We believe that it is possible to add WS-Agreement compliance into a system built on NDF once the specification of WS-Agreement is finalised.

Although we have determined that NDF is a suitable basis for negotiating over QoS, it does not provide the features we require in order to support negotiation with intermediaries, in order that a *distributed* notification service can make use of shared subscriptions from pre-existing negotiations. If multiple consumers at a notification service request subscriptions to the same topic, the notification service can recognise this and share a subscription for all of them. However, if the consumers have requested different QoS levels from each other, this may not be possible. A solution is required where the negotiation process not only takes into account the preferences of the client and supplier in the negotiation, but also any existing commitments held by intermediaries between them.

These intermediaries could offer to satisfy the client's request without any further intervention from the supplier, reselling or redistributing the item already being received. No existing models of negotiation support this pattern of negotiation. Hence we propose *chained negotiation* as a model supporting the redistribution or reselling of items or services obtained previously for other clients. In a distributed notification service, this will enable subscriptions to notifications to be shared between multiple consumers. We discuss this further in Chapter 5.

Chapter 4

Direct Negotiation Engine

In Chapter 3 we introduced the concept of negotiation, and discussed existing work in the field of negotiation. In this chapter, we introduce a direct negotiation engine: a component that can automatically negotiate over a set of conditions on behalf of another system (the host), with minimal interaction required from the host.

4.1 Introduction

Negotiation is the process by which two or more parties attempt to reach a mutually acceptable set of terms over which an item or a service should be exchanged. It can enable differences in preferences about how a service is delivered to be resolved. However, since negotiation can be a complicated process, it is desirable for the negotiation behaviour to be encapsulated in a reusable component, allowing negotiation to be supported by many different applications in a standard manner without those applications needing to be aware of *how* to negotiate. We have designed DiNE, a direct negotiation engine based on bilateral negotiation which enables external services to support negotiation.

DiNE is intended for use in a notification service, where consumers subscribe to a publisher to receive notifications on a particular topic, while being able to specify some Quality of Service constraints and have DiNE automatically resolve differences in the requirements of the publisher and consumer.

The novel contributions presented in this chapter consist of: an empirical evaluation of a heuristic-based automated negotiation model showing behaviour not previously demonstrated, giving us a better understanding of the model; and a demonstration of the suitability of the negotiation model for resolving differences between QoS preferences in a notification service. While existing notification services allow consumers to specify QoS conditions with a subscription, they do not allow differences in the preferences of a service provider and consumer to be resolved.

Later in this thesis we will extend this contribution to include intermediaries in a negotiation (Chapter 5) and to integrate this work with a distributed notification service, enabling service providers to automatically negotiate over QoS with consumers, and allowing distributed notification services to share subscriptions to the service provider, further increasing the number of consumers that can be satisfied without overloading the service provider.

The rest of this chapter is organised as follows: In Section 4.2, we present the design of DiNE, our direct negotiation engine, giving a detailed description of the protocols and strategies employed. We then evaluate the behaviour and performance of our negotiation engine using an experimental evaluation in Section 4.3. In Section 4.4, a scenario is presented where the negotiation engine is integrated into a notification service publishing bioinformatics data. We show that using negotiation in this scenario can reduce the load requirements on the service provider, enabling more clients to be serviced. Finally in Section 4.5, we summarise this chapter.

4.2 Negotiation Engine Design

In this section we present the design of DiNE, our Direct Negotiation Engine. DiNE encapsulates the protocol required to perform negotiation, enabling external systems to support negotiation, without needing to know how to negotiate. These systems may still influence the negotiation process at the strategy level, for example, in the form of resource levels for use in utility functions and proposal generation.

DiNE is based around the Negotiation Decision Functions model of negotiation, introduced previously Section 3.3.2. As this is a fundamental part of DiNE, we expand on this below before discussing DiNE.

4.2.1 Negotiation Decision Functions

Negotiation Decision Functions (NDF) is an approach to automated negotiation using a heuristic-based approach, and is presented by Faratin (2000) as well as numerous other papers (Sierra et al., 1997a; Faratin et al., 1997, 1999a,b, 2000). Faratin introduces a bilateral (one-to-one) negotiation model which clearly separates negotiation protocol and strategies. This enables many different negotiation strategies to be used, without having to change the protocol. Agents using this model simply supply the resource functions and make decisions that will influence the outcome of the negotiation, enabling them to effectively negotiate with varying levels of understanding of required protocols and strategies. The routines for proposal generation and evaluation can be chosen from a range of predefined functions, some of which make use of an external resource function, measuring the availability of some system resource. Throughout this thesis, we will refer

to this model as *Negotiation Decision Functions* (NDF). A more complete description of NDF is presented in Faratin (2000).

In NDF, the subject of a negotiation is a *contract*, representing the current or final bid in the negotiation, and contains values for one or more *issues*, the values that will change over the course of a negotiation. A negotiation is an alternating sequence of offers and counter-offers by a client to obtain the item or service referenced in the contract, terminating in either a commitment by both parties to a mutually agreed solution, or a failure to reach such an agreement (unsuccessful termination). To negotiate with each other, agents must have a shared understanding of the individual issues before the negotiation starts. A common way of sharing this understanding is by use of an *ontology*, which is a specification of a concept (Gruber, 1993).

Each agent has preferences defining the limits of a continuous range of values each issue is permitted to take in order to be considered acceptable. In this context, acceptable values are those that an agent would be prepared to commit to. It does not mean the agent will take this course of action — it may try and obtain a better offer to accept if there is still time remaining for a negotiation to be completed. For quantitative issues, preferences define a range of values considered acceptable by the agent for each issue. Qualitative issues are more complicated, as they do not use a continuous set of values. Instead, the model imposes the restriction that the discrete values a qualitative issue can take must be defined over an ordered domain, and the limits of the preferences are redefined as the limits of that issue's score.

To evaluate offers, each agent has a scoring function that takes a value for a single issue and returns a score between 0 and 1. An agent also assigns a relative importance weight to each issue. These scoring functions and weights are combined using a weighted average function to give an overall score for the contract.

It is assumed that the two parties involved in a negotiation will have conflicting interests — for example a buyer normally wants to obtain something as cheaply as possible, while the seller strives to maximise the amount of money they would receive. This allows each agent to assume it will know the direction of any change in preferences of the opponent, although it will not know the exact details of their preferences. Additionally, the assumption is made that scoring functions are either monotonically increasing or decreasing.

Proposals are generated using methods based on *tactics* and *strategies*, as follows. Tactics are functions that generate a value for a single issue for inclusion in a proposal based on a single criterion, such as the remaining time or available system resources. Three types of tactic are defined: *time-dependent* tactics use the amount of time remaining until the deadline to concede; *resource-dependent* tactics use a resource function in a similar way to the remaining time, conceding based on the availability of a specified resource; and *behaviour-dependent* tactics react to the offers received from their opponent.

Examples of the resources used by resource-dependent tactics include the current system load and the number of negotiations currently in progress. This typically involves a callback to a resource function external to the negotiating agent.

Tactics are combined using weightings to produce values for proposals that may be based on more than a single criterion, and agents have the ability to vary these weightings over the course of a negotiation. For example, a resource-dependent tactic may be weighted more highly at the start of a negotiation, with the bias switching to a time-dependent tactic as the deadline approaches. This behaviour is the agent's *strategy*. Strategies are used to combine tactics instead of generating proposals purely as a result of a single factor.

Further work on NDF has enabled trade-offs to be calculated (Faratin et al., 2000), enabling better counter-proposals to be generated that are more likely to be acceptable by the other party. Faratin (2000) also discusses an extension to enable dynamic issue set manipulation, where issues can be added or removed from a negotiation while it is still running. This can enable a negotiation that has become deadlocked (where neither party are willing to make further concessions) to continue (by removing resolved issues, or introducing a new one to broaden the range of possible outcomes).

This model was chosen for the following reasons: it enables the system to be designed in such a way that the negotiation protocol can be completely handled by DiNE, while allowing other systems using DiNE to influence the negotiation strategy; and it enables negotiation to take place without requiring additional third parties to participate in the negotiation process; some automated negotiation models such as Bartolini et al. (2002) support a more general form of negotiation, but require a third party to host the negotiation process. In a distributed environment such as the notification service scenario we are using, a dedicated negotiation host would lead to performance and/or availability problems under load, as many services try and make use of it.

4.2.2 Core Concepts

In this section we introduce the concepts which will be used throughout the rest of this chapter and subsequent chapters discussing DiNE. A negotiation occurs between a *client* and a *supplier*, where a client tries to obtain a *negotiation item*[1] from the supplier, which could be goods or a service. When applied to the idea of a notification service, the client is a consumer negotiating for a subscription to notifications on a particular topic (the negotiation item) from the publisher (the supplier). Our terminology differentiates between clients and consumers, and publishers and suppliers, to indicate where we discuss

[1] A negotiation item is referred to as a contract in NDF — we use the term negotiation item to indicate a broader use for DiNE than for contract negotiation, being suitable of the acquisition of goods or services.

concepts specific to notification services and to negotiation in general. The client and supplier are both *hosts* to the negotiation engine.

The values that are under negotiation are attributes of the negotiation item or the way in which the item would be delivered. These are referred to as *issues* (which correspond to *agreement contexts* in WS-Agreement (Andrieux et al., 2004)). For example, consider the use of negotiation in a notification service. A consumer would try to set up a subscription for a particular topic, and would negotiate over various Quality of Service parameters, such as: message size; accuracy; granularity of notifications; duration of subscription; and cost.

A conversation between the client and supplier, where proposals and counter-proposals are exchanged, is known as a *negotiation thread*. Both the client and supplier have *preferences*, representing the *ideal value* (i.e. the value they would like to obtain in an ideal world) and a *reservation value*, placing a limit on the concessions that will be made. Values beyond the reservation value are unacceptable to that party, and a negotiation will fail if it is not possible to reach a proposal that satisfies the preferences. Each party measures the *utility* of an outcome — the measure of how good the outcome is for that party. It is rational behaviour for an agent to try and maximise their expected utility (Simon, 1955).

In the process of negotiation, each party tries to maximise their *utility* — the measure of how good an outcome is for a particular party.

4.2.3 Architecture

At a conceptual level, DiNE can be regarded as an entity shared between all of the participants in a negotiation, as shown in Figure 4.1. The client and the supplier use some shared entity to negotiate on behalf of them, resulting in a mutually acceptable proposal to which they can commit. To realise this concept, there are two possible architectures of DiNE: 1) A shared negotiation engine mirroring the conceptual architecture; and 2) a negotiation engine using separate *negotiation components* linked to each party.

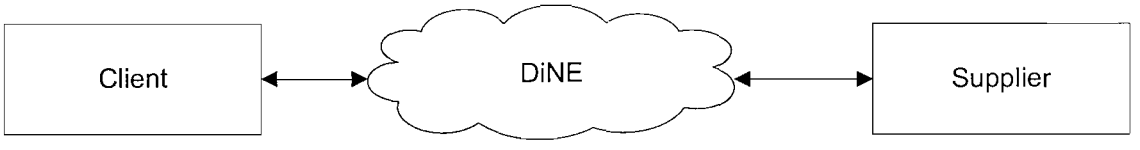

FIGURE 4.1: Conceptual Architecture of DiNE

4.2.3.1 Shared Negotiation Engine

A single negotiation engine shared between both the client and the supplier takes the preferences from both parties, carries out all of the negotiation internally, and informs each party of the outcome. Neither party needs to be aware of any negotiation protocols in use — they just need to know that conflicts between their preferences are being mediated externally. In order for the negotiation engine to be able to handle the negotiation, it needs to know the preferences of both the client and the supplier in a negotiation. It also needs any information from each party that would influence the decision making process within the negotiation. Assuming this is based in a service-based architecture, it is possible that the client, supplier and negotiation engine are all located in different places.

A shared negotiation service design enables the easy provision of negotiation between the involved parties without either of them needing to know the details of how the negotiation works. All aspects of the negotiation can be hidden from the hosts, enabling them to simply request an agreement, and receive back either an agreement or a failure.

However, there are two main disadvantages with the shared negotiation engine architecture. Certain aspects of a negotiation, including how to evaluate proposals, may require the use of dynamic, locally-sensed environmental values from the hosts, such as the system load or amount of free disk space. This either requires the negotiation engine be co-located with the host and has access to these variables, or it requires the host to provide a callback mechanism for the centrally-hosted negotiation engine to obtain these values. Co-locating the negotiation engine with one party (e.g. the supplier) makes it harder for the other party (client) to make use of this facility, as it would rely on the client trusting the supplier. Providing a callback mechanism does enable the negotiation engine to access these variables, but at the expense of the simplicity of the shared architecture.

The second disadvantage of this architecture is that the negotiation engine has access to the preferences and other information of both parties in the negotiation, thus requiring that both parties implicitly trust the same negotiation engine. For example, if the main aspect of a negotiation was the price to be paid for a service, a malicious negotiation engine could tell each party that the negotiation completed at their reservation values, while keeping the difference between values for itself. Similarly, where the negotiation engine is colluding with the supplier, an outcome could be obtained that is only just acceptable for the client, while proving extremely favourable to the supplier.

4.2.3.2 Separate Negotiation Components

In the second proposed architecture of DiNE, each party in the negotiation hosts its own *negotiation component* (NC), which tries to find a mutually acceptable proposal

over which an agreement may be formed on behalf of its host. This is shown in Figure 4.2. The NCs are given the preferences of their hosts at the start, and are able to request information from their host throughout the process of the negotiation for use in counter-proposal generation and evaluation. Thus, locally-sensed environmental values can be easily integrated as factors in the negotiation.

FIGURE 4.2: Separate negotiation component architecture

This architecture enables the use of inputs from the host to influence or control the process of how proposals are generated and evaluated, while still allowing the host to remain ignorant of the negotiation protocol. As the NC is private to each host, there are no issues regarding trust between the host and NC. This enables interactions in untrusted environments — if a negotiation component finds a mutually acceptable proposal, it is within the limits of what the host has requested from its NC. With this architecture is that the host can vary the degree to which it influences the outcome of the negotiation anywhere between simply supplying the preferences, and supplying custom routines for proposal evaluation and generation.

The disadvantage of separate NCs is that the interaction pattern is more complex. With a shared negotiation engine, each party instructs the negotiation engine to find a mutually acceptable proposal, and no more interactions occur until the negotiation has terminated. With separate NCs, interactions occur between a host and its NC, and between the two NCs. If the negotiation engine is to be independent of a particular communication system, the host must facilitate communication between the two NCs by way of implementing a message transport.

4.2.3.3 Chosen Negotiation Engine Architecture

The architecture of DiNE is based on the approach of using separate NCs discussed in Section 4.2.3.2 above, so that external locally-sensed environmental variables could be used in the negotiation, and so that the privacy of the preferences and utility functions of the hosts can be maintained even in an untrusted environment. This architecture allows the hosts a varying level of control over the negotiation process, by allowing them to choose between the provided routines for proposal generation and evaluation and their own modified versions of the routines. The NCs use their host's communication mechanism to send proposals so that the design of the engine remains independent of a communication mechanism (for example, in Section 6.2.3, we describe an implementation

based around a Web Services model). The alternative proposal suggested in Section 4.2.3.1 is rejected because of the requirement that both parties need to trust the same negotiation engine, and because hosts cannot influence the negotiation process as easily as with separate NCs.

Figure 4.3 shows in more detail the parts that make up the architecture of DiNE, showing the sections that can be shared with the host. The main part of the negotiation protocol handling is done by the NC internally. Proposal generation and evaluation is controlled by the NC. However, hosts can supply their own routines for proposal generation and evaluation which will be used by the NC. Additionally, the routines may call for information to be provided by the host, such as information about system resources. The host is required to provide a *reliable* message transport for use by the NCs. Communication is the only part of this architecture that must involve the host.

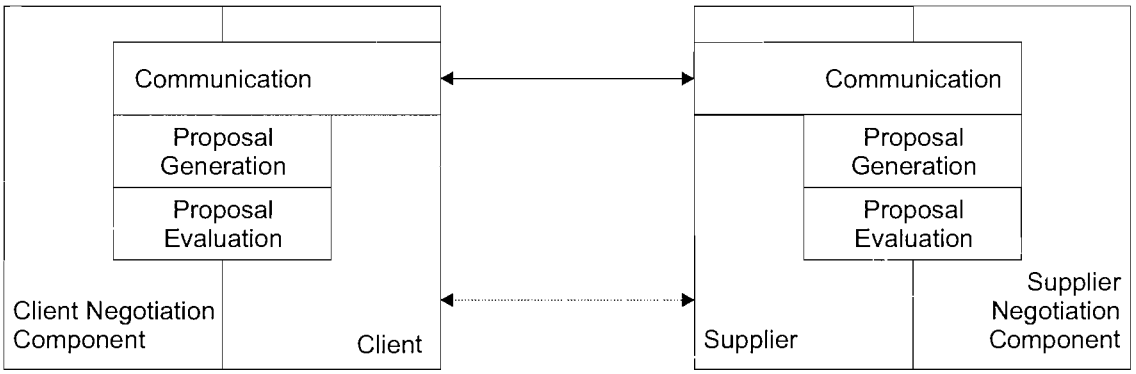

FIGURE 4.3: Detailed Architecture of DiNE

4.2.4 Negotiation Protocol

The negotiation protocol specifies how each party in a negotiation component can act: the messages that can be sent, the actions that can be taken and the various states the negotiation process can take. A clear specification of this information is essential to understanding the interactions that can occur between the various parties involved in a negotiation, and how these interactions are interpreted.

4.2.4.1 Interactions in a negotiation

Figure 4.4 shows a negotiation taking place between a client and a supplier through their respective NCs. The interactions between each party are shown as arrows with the message type. The numbers indicate the stages of the negotiation, which are explained below.

1. Preference Registration

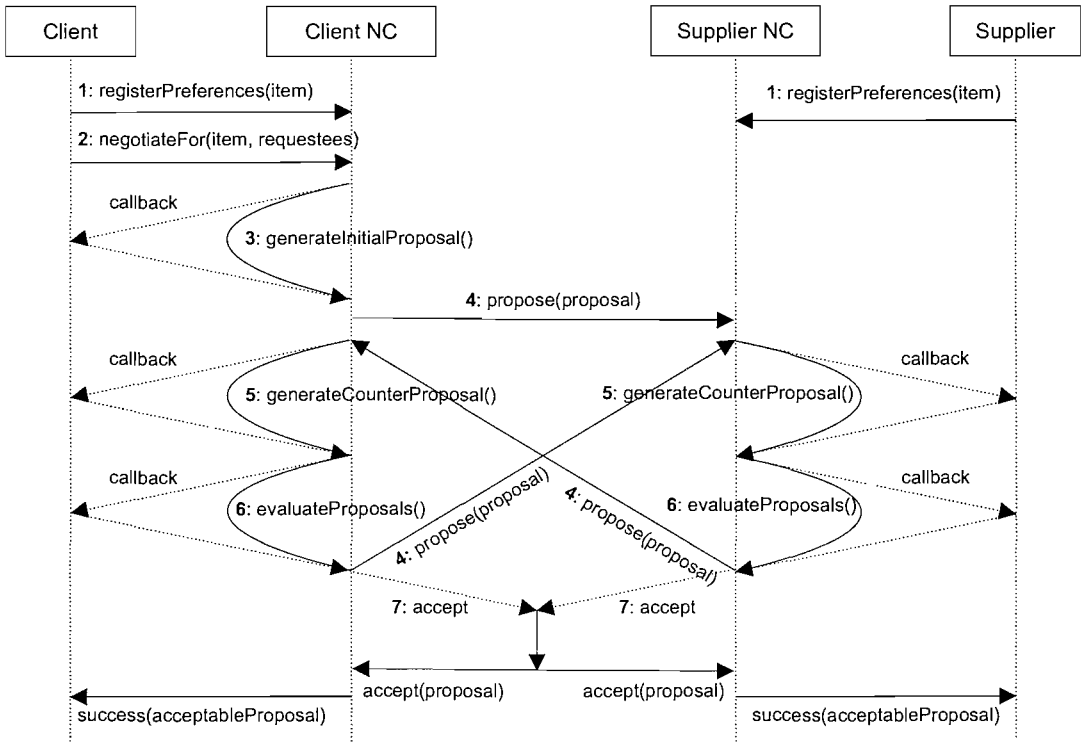

FIGURE 4.4: Interactions between components of a negotiation

Before the negotiation begins, both parties initialise their respective negotiation component with a set of preferences for the items they are interested in or are making available. The preferences consist of two values, an ideal value and a reservation value which, combined, represent the range considered acceptable for a specific issue.

2. Negotiation Initiation

   The client gives its negotiation component an instruction to begin the negotiation process.[2] In order to constrain the amount of time a negotiation can take, a deadline is specified as the negotiation is started. Not only does this ensure that a negotiation will complete, it also ensures that if the item is required in a specific time frame, the negotiation will have completed (with either a success or failure) within this time frame.

3. Initial Proposal Generation

   The client's negotiation component generates an initial proposal ready to send to the supplier's negotiation component. Typically, this would contain the ideal values for all of the issues involved in the negotiation. Optionally, this step can involve callbacks to the client in order to give the client a chance to influence the initial proposal generation.

[2]Although it is possible for the supplier to initiate a negotiation, we concentrate on the more typical case where a client initiates the negotiation (for example, a customer tries to buy something from a seller, or requests a service from a service provider). If the supplier were to initiate the negotiation, the diagram would look identical apart from stage 2 coming from the supplier to the supplier's NC.

4. Proposal Transmission

   The negotiation process now enters a cycle, initiated by a proposal being transmitted. Negotiation messages are transmitted to their recipient by the host. The NCs assume that the communication mechanism is reliable — there is no mechanism for dealing with missing messages.

5. Counter-proposal generation

   When a proposal is received by a NC, a counter-proposal is generated. Detailed information on proposal generation can be found in Section 4.2.5.3. As with initial proposal generation, counter-proposal generation can involve callbacks to the host, in order to retrieve external environmental conditions such as system load, or to enable the host to provide a customised proposal generation routine.

6. Proposal Evaluation

   Once the counter-proposal is generated, both the generated counter-proposal and the recently received proposal are evaluated using p's utility functions. A comparison is made between the score of the counter-proposal and the received proposal. If the counter-proposal has a higher utility, it is transmitted to the other negotiation component, and the process returns to step 4. If the received proposal has equal or higher utility than the generated counter-proposal, it is considered acceptable.

7. Proposal Acceptance

   If the last received proposal has a utility equal to or higher than that of the generated counter-proposal, an acceptable state has been reached. The negotiation component issues an Accept message to signify its acceptance of the received proposal. Both negotiation components then inform their respective hosts that an acceptable state has been reached.

Acceptance of a proposal by the negotiation components does not constitute a *commitment* between the client and the supplier — it merely indicates that both parties have found a mutually acceptable set of issues to base a commitment on. The client would typically contact the supplier after a successful negotiation to make a commitment — a contract where the supplier agrees to supply the client with the negotiation item under the issues agreed on during the negotiation. Leaving the process of making a commitment to the hosts enables a client to negotiate with multiple suppliers simultaneously, and to proceed to making a commitment with the one that offered the best deal.

The diagram in Figure 4.4 only shows the case where a negotiation is successful, but negotiations can fail for two main reasons. The most likely reason for a failure would be deadline expiry — no mutually acceptable proposal is found before the deadline expires. In this situation, no counter-proposal would be generated on receipt of a proposal if the deadline has expired. Instead, a negotiation failure message would be sent giving the

reason for the failure. Negotiations can also fail due to being explicitly terminated by either the client or supplier, for example when a system is being shut down, if the client has found another supplier for the item being negotiated for.

The negotiation protocol described above relies on a reliable communication protocol — it makes no allowances for messages not being delivered. Additionally, the interactions between a NC and its host are synchronous — when the client begins a negotiation, its NC returns the negotiation outcome. Similarly, if the NC requires a callback to the host, for example to obtain a resource level, the NC will wait for the host to provide this information before continuing with the negotiation. There is no specification of how the communication between the NCs should happen — it is left to the NCs to implement. This allows the negotiation engine to be integrated into a number of different distributed environments — we integrate it with a SOAP messaging system later in this thesis (Section 6.2.3).

4.2.4.2 Message Types

While trying to reach an agreement over a particular item, negotiation components exchange messages to convey their current offers and reactions to the offers. Three types of messages are used:

- Propose

  Propose messages are the main part of any negotiation. They convey an offer from one negotiation component to another with a set of values proposed by the sender for each of the issues in the negotiation.

- Accept

  Accept messages are used when a negotiation component receives a proposal that it is prepared to accept. An Accept message simply contains a reference to the proposal it is related to.

- Terminate

  Terminate messages are used to signify a negotiation failure or explicit termination. A reason field is contained in the message to indicate to the other party why the negotiation failed. The most common reason for failure would be deadline expiry.

All messages contain a common message header for identifying the message type, source and destination of the message, and for identifying the negotiation the message is related to. Details of the fields found in the message header, as well as the fields in the specific messages, are detailed in Table 4.1.

Message	Field Name	Description
(All)	MessageType	Identifies the type of this negotiation message
	SenderID	Identifier for the sender of this message
	RecipientID	Identifier for the recipient of this message
	NegotiationID	Unique identifier for this negotiation
Propose	ProposalID	Unique identifier for this proposal in the negotiation
	Item	Item this negotiation is for
	Elements	List of values for issues in this negotiation
	TimeRemaining	Amount of time remaining in this negotiation
Accept	ProposalID	ID of the proposal being accepted
	TimeRemaining	Amount of time remaining in this negotiation
Terminate	Reason	Reason for the negotiation being terminated

TABLE 4.1: Negotiation Message Structure

A sequence of messages between two parties regarding one particular negotiation is known as a negotiation *thread*. Threads are started using a propose message, and terminated using an accept or terminate message. The thread is made up of an exchange of propose messages, and is terminated either by an Accept message or a terminate message.

4.2.5 Negotiation Strategy

The protocol described in the previous section defines the rules which must be adhered to when negotiating using this model. However, there are many possible courses of action that may be followed while conforming to this protocol. The negotiation strategy describes the type of actions that a negotiation component performs within the protocol. There is no requirement for the negotiation strategy to be fixed or the same between opponents — it is possible for two NCs with completely different strategies to negotiate under the same protocol.

The negotiation strategy used in DiNE is controlled by a set of functions for influencing proposal generation and evaluation routines. It is up to the host to select these functions — predefined routines may be used, or the host can supply their own routines. The functions are used by the negotiation algorithm to enable the host to remain ignorant of the negotiation protocol, while maintaining control over the behaviour of the NC. The functions generate values for inclusion in proposals, and to evaluate proposals and counter-proposals.

4.2.5.1 Terminology

Through the rest of this section we will be detailing the components used in the strategy part of the NC. We introduce here some terminology and notation that will be used to

describe the different parts.

$$\begin{aligned} \mathcal{C} &= \{c_0, c_1, \ldots\} && \text{(Set of clients)} \\ \mathcal{S} &= \{s_0, s_1, \ldots\} && \text{(Set of suppliers)} \\ \mathcal{X} &= \{x_0, x_1, \ldots\} && \text{(Set of negotiation items)} \\ n & && \text{(Number of negotiation issues)} \\ \mathcal{Q} &= \{q_0, q_1, \ldots, q_n\} && \text{(Set of } n \text{ negotiation issues)} \\ t_{max} & && \text{(Deadline at negotiation start)} \end{aligned}$$

Characteristic variables are:

$$c \in \mathcal{C}; s \in \mathcal{S}; a, b \in (\mathcal{C} \cup \mathcal{S}); x \in \mathcal{X}; q \in \mathcal{Q}$$

A *proposal* sent from *src* to *dest* is represented as a tuple:

$$p = \langle src, dest, id_{neg}, id_p, t_{rem}, x, E \rangle \, .$$

To relate all messages to a specific negotiation, a unique identifier id_{neg} is used to identify the negotiation to which they belong. This identifier is created by the client initiating the negotiation. A proposal also contains a proposal identifier, id_p, which is unique within the negotiation, and which is used in Accept messages to identify the proposal being referenced. The set of proposal elements E is a set of values included in a single proposal specifying a value for each issue under negotiation. The notation $p_{src \to dest}$ is used as shorthand for specifying the source and destination of the proposal, and the notation $p[q]$ represents the value of element q from within proposal p.

An example proposal between client a and supplier b for item *item01* might contain the following information:

$$p = \langle a, b, \text{'neg001'}, \text{'prop-005'}, 10, \text{'item01'}, (\text{price=0.2,timeliness=5}) \rangle$$

A negotiation thread $\mathcal{P}$ is an ordered list of proposals exchanged between two negotiation parties a and b at time t ($t \leq t_{max}$):

$$\mathcal{P}^t_{a \leftrightarrow b} = \left(p^0_{a \to b}, p^1_{b \to a}, \ldots, p^t_{a \to b}\right) .$$

It represents the history of a negotiation from the receipt or transmission of the first message, up to the most recent.

An Accept message sent between *src* and *dest* is represented by

$$acc = \langle src, dest, id_{neg}, id_p, t_{rem} \rangle \, ,$$

where id_p represents the proposal identifier of the proposal being accepted. If a negotiation is unsuccessful, it is terminated using a failure message, represented as

$$fail = \langle src, dest, id_{neg}, reason \rangle$$

where *reason* indicates the reason for failure.

4.2.5.2 Proposal Evaluation

The NCs rely on being able to evaluate a proposal to determine how useful it is to the host. The scoring function mechanism enables a negotiation component to create a mapping between values for each issue to a *utility* for its host.

Preferences given by a host to its negotiation component for each issue comprise two values — the ideal and reservation value. The ideal value gives the highest utility for that party, i.e. the value they would like to receive. The reservation value represents the absolute limit on any concessions, i.e. the value giving the lowest possible utility while still being acceptable. Values beyond the reservation value are not accepted.

Issues can be *quantitative* or *qualitative*. The range of values considered acceptable by party a for issue q is represented as $D_q^a = [ideal_q^a, res_q^a]$ for a quantitative issue. Qualitative issues must be handled differently; the range of values a qualitative issue can take must be defined as an ordered set, with scores being assigned to each one. The values for $ideal_q^a$ and res_q^a can be defined as the values giving the maximum and minimum score for the issue, allowing qualitative issues to be treated similarly to quantitative issues.

To determine the valuation placed on the value for a single negotiation issue q, party a has a scoring function $v_q^a : D_q^a \rightarrow [0, 1]$. The only requirement of a scoring function is that it be monotonically increasing or decreasing. Within these limitations, the scoring function could be linear between the preference limits, or it could take any other shape such as exponential.

Scoring functions produce a score for a single issue only. A weighted summation function is used to combine these scoring functions with their weightings to give an overall *proposal scoring function*:

$$V^a(p) = \sum_{i=1}^{q_n} w_i^a v_i^a(p[q_i])$$

4.2.5.3 Proposal Generation

Proposals are generated using *tactics* — functions that generate a value for a single issue based on a single criterion, such as the time remaining before a deadline, or avail-

able system resources. As tactics only use a single criterion to generate a value, a NC would normally use a combination of different tactics. Three families of tactic were introduced in NDF: time-dependent, resource-dependent and behaviour-dependent, which are summarised below

Time-dependent Tactics

The basic form of tactic is one whose concession rate is controlled exclusively by the amount of time remaining in the negotiation: time-dependent tactics. These tactics do not take into account any information about the local environment. However, they are still important as they influence a concession in a negotiation; negotiations that do not concede are unlikely to reach a successful proposal. Time-dependent tactics generate a value for each issue based on the amount of time remaining using the following expression:

$$p[q] = ideal_q + (1 - \alpha_q(t))(res_q - ideal_q)$$

The definition of the function α depends on the type of tactic:

- **Polynomial:** $\alpha_q(t) = \kappa_q + (1 - \kappa_q)(\frac{min(t, t_{max})}{t_{max}})^{\frac{1}{\beta_q}}$
- **Exponential:** $\alpha_q(t) = e^{(1 - \frac{min(t, t_{max})}{t_{max}})^{\beta_q} ln \kappa_q}$

There are two controlling parameters used as input to the α function — κ and β. The initial offer made is controlled by κ_q : when $\kappa_q = 0$, the initial offer is the ideal value for that issue, and with $\kappa_q = 1$, the initial offer is the reservation value. The rate of concession is controlled by the parameter β. Different values for β cause the values produced to concede in different patterns. The behaviour of both exponential and polynomial time-dependent tactics can be seen in Figure 4.5.

The behaviour of time-dependent tactics can be classified into three different families: Boulware, linear and conceder (Faratin, 2000). We will explain these below, using values for β from polynomial time-dependent tactics, as exponential time-dependent tactics do not exhibit a true linear behaviour.

Boulware tactics ($\beta < 1$) concede very slowly for most of the negotiation. As the deadline approaches, the concession rate increases dramatically towards the reservation value. Boulware tactics aim to keep the utility higher for those using it, but this behaviour comes at the risk of not making a deal at all.

Linear tactics ($\beta = 1$) are the simplest of all time-dependent tactics, and concede at a consistent, predictable rate throughout the course of the negotiation.

Conceder tactics ($\beta > 1$) are suitable for occasions where an agreement must be reached very quickly. The conceder tactic drops close to its reservation value very early on in

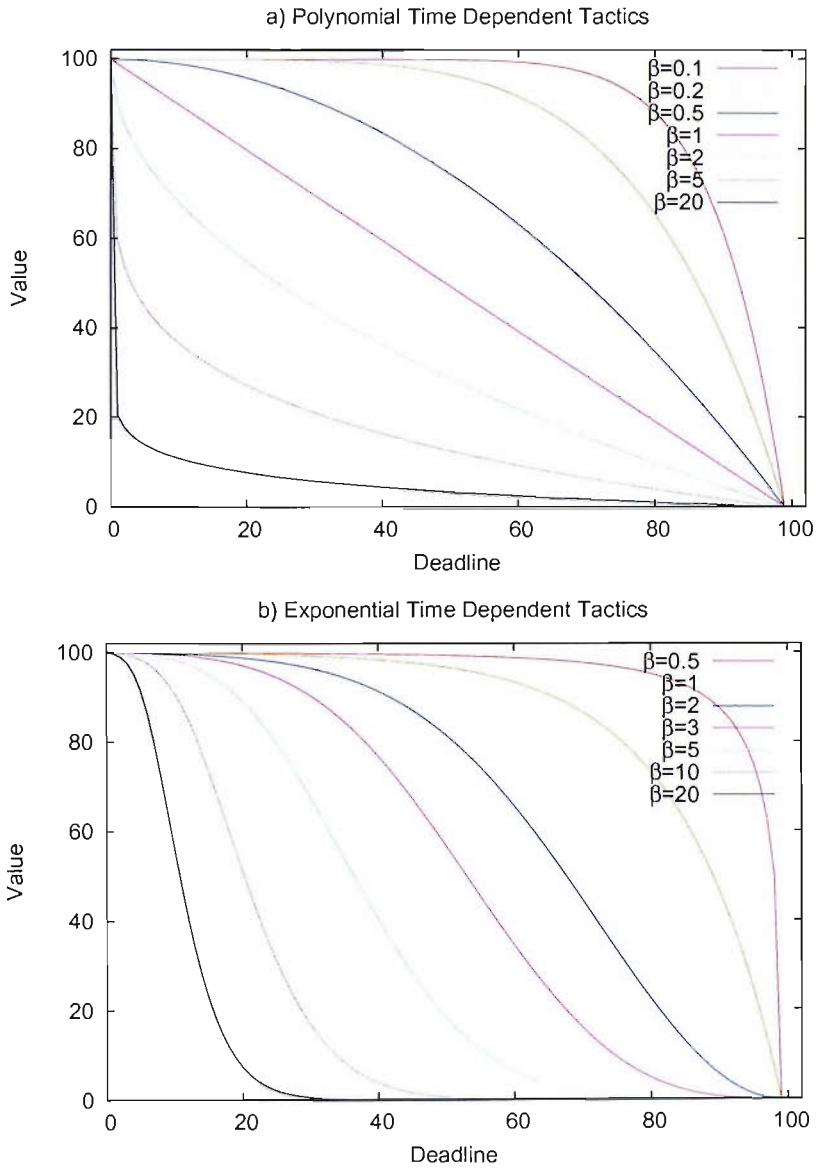

FIGURE 4.5: Behaviour of a) Polynomial and b) Exponential time-dependent tactics

the negotiation, with the rate of concession dropping as time passes. After the initial concession it reaches its final reservation value much more gradually.

Resource-dependent Tactics

Time-dependent tactics are useful for ensuring that some concession does occur in a negotiation, but does not take any environmental conditions into account. For this purpose, resource-dependent tactics should be used. Resource-dependent tactics need a mechanism for obtaining resource levels from the host to be used as the controlling factor for generating a value to be returned. Examples of these resource levels could be system load or the number of commitments already in place.

Behaviour-dependent Tactics

Behaviour-dependent tactics calculate the next value for an issue based on the behaviour of their opponent. Proposals are generated based on the change between proposals received from the opponent. This change can be imitated relatively, absolutely or using an average of the changes. However, Axelrod has established that behaviour-dependent tactics can never do better than other tactics; they can only gain equal utility to the best tactic (Axelrod, 1984). Hence, although behaviour-dependent tactics are supported by DiNE, we have chosen to omit them from any evaluation.

Tactic Combinations

A NC typically uses a number of different tactics based on the constraints they are negotiating in. Multiple tactics are combined using a weighted average in order to give preference to different criteria for proposal generation. It is possible to vary these weightings at any time during the course of the negotiation. *Strategies* control the weightings of the individual tactics, and how they are varied over time. A strategy could, for example, increase the weighting against a time-dependent tactic as the deadline approaches to ensure that a negotiation concedes at the end.

4.2.5.4 Negotiation Process

In order for a client to begin a negotiation with a supplier, it must have first determined the identity and the item for which negotiation will take place. Selection of negotiation partners is beyond the scope of the negotiation process and is assumed to have been carried out by the host prior to a negotiation being initiated, possibly using a directory service, such as UDDI.

Algorithm 1 shows that to begin a negotiation, a client NC first assigns a unique identifier to the negotiation. This is attached to all messages in the negotiation to link them together. The NC then generates a set of values for the initial proposal using the tactics and preferences (described in Section 4.2.5.3). The proposal is then constructed and sent using the host's communication mechanism.

Algorithm 1 Client c initiating a negotiation with s for item x, with max time t_{max}

id_{neg} = genUID()
id_p = genUID()
$\mathcal{P} = ()$
E = applyTactics(t_{max}) {Generates values for a proposal by applying tactics}
$p^0_{c \to s} = \langle id_{neg}, id_p, c, s, t_{max}, x, E \rangle$
sendMessage($p^0_{c \to s}$)
$\mathcal{P} = \mathcal{P} \cup \{p^0_{c \to s}\}$

When messages are received, they are handled by Algorithm 2. When a message is received, a counter-proposal is immediately generated using the combination of tactics. Both the received and generated proposals are evaluated using the proposal scoring

functions (described in Section 4.2.5.2). If the value of the incoming proposal is higher or equal to that of the received proposal, an Accept message is sent and the negotiation component's host is notified of the successful negotiation. Otherwise, the deadline is examined. If there is time remaining, the generated proposal is sent using the host's communication mechanism. Otherwise, a failure message is sent and the host is notified that no agreement was made.

Algorithm 2 Negotiation Component a processing a proposal message from b

p_{recv} = receiveProposal()
$\langle id_{neg}, id_{p_{recv}}, b, a, t, x, E_{recv} \rangle = p_{recv}$
E = applyTactics$(t-1)$ {Generates values for a proposal by applying tactics}
$id_{p_{new}}$ = genUID()
$p_{new} = \langle id_{neg}, id_{p_{new}}, a, b, t-1, x, E \rangle$
if $V^a(p_{recv}) \geq V^a(p_{new})$ **then**
  sendMessage($\langle id_{neg}, id_{p_{recv}}, a, b, t-1 \rangle$) {Send an Accept message}
  notifyHost(success, p_{recv}) {Inform host of the success}
else
  if t = 0 **then**
    sendMessage($\langle id_{neg}, a, b,$ 'Deadline Expiry'$\rangle$)
    notifyHost(failure, 'Deadline Expiry') {Inform host of failure with reason}
  else
    sendMessage(p_{new}) {Send generated proposal}
  end if
end if

4.3 Experimental Evaluation

DiNE has been designed to incorporate a number of external inputs as controls over the negotiation process. As such, it is not possible to undertake an experimental evaluation covering the full range of situations in which it could be used. In this section, we evaluate the performance and behaviour of DiNE while using only time-dependent tactics and linear proposal evaluation routines. Observing the system under these conditions gives an indication of the behaviour without any external influences, showing any behaviour patterns which may influence the combination of other tactics.

In this evaluation, we focus on time-dependent tactics, as these are likely to remain the same in a working implementation due to no reliance on resources. Time-dependent tactics force a negotiation to concede towards the deadline, and are useful for ensuring that concessions are made. We do not expect time-dependent tactics to be used alone in a real implementation; resource-dependent tactics would be used to factor in environmental conditions, and behaviour-dependent tactics may be used to react to the behaviour of the opponents. However, resource-dependent tactics are domain-specific by their nature — they rely on an environmental factor to control any concessions made, making it impossible to evaluate them independently of specific resources.

To evaluate the behaviour of DiNE, three experiments were run: in the first experiment we varied the amount of time available in which to negotiate; in the second experiment, the number of issues being negotiated over was varied, and the final experiment measured the performance of our implementation in terms of real time.

4.3.1 Experiment Setup

The experiments in this section all share the same basic structure. A set of *environments* is generated containing the variables that restrict the outcome of each negotiation, namely the preferences and deadline (described below). NCs run a negotiation in each environment using each tactic. For every tactic, a NC negotiates against every other tactic (including itself) for each environment in the experiment. The outcomes of the negotiations were collated and averaged.

4.3.1.1 Environments

The variables that control the outcome of a negotiation are primarily the time available in which to negotiate, and the preferences of each party for each of the issues. We group these variables into an *environment* for the purpose of analysis. Since the space of possible issues is not defined, the number of potential environments is infinitely large. For our experiments, we created a set of randomised environments that can be used for repeatable experiments.

For each environment, there are a number of issues defined (depending on how many are required for the experiment). For each of these issues, the minimum and maximum acceptable values for each party are defined by the following variables: $ideal_q^c$ representing the ideal value for c for issue q, θ_q^c controlling the size of the range of acceptable values for c, θ_q^s controlling the size of the range of acceptable values for s, and ϕ_q, the fraction of the acceptable regions that overlap. If $\phi_q = 0$, there is no overlap between the acceptable regions of the preferences, and a successful outcome cannot be found. If $\phi_q = 1$, both parties have fully-overlapping preferences, and any value acceptable to one party is also acceptable to the other. These parameters are shown in Figure 4.6, giving an overview of how they produce the remaining preference values. Table 4.2 shows the ranges of values used when generating proposals.

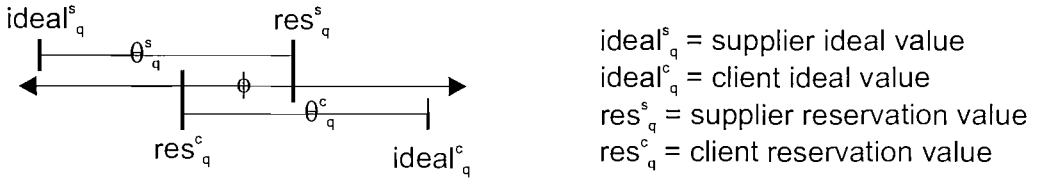

FIGURE 4.6: Effect of environment parameters on preferences for issue q

Parameter	Explanation	Range
$ideal_q^c$	Ideal value for client	30
θ_q^c, θ_q^s	Size of acceptable regions	10–50
ϕ_q	Amount over overlap in acceptable regions	0–0.99
t_{max}	Amount of time available in which to negotiate	10–60

TABLE 4.2: Values used for environment generation

We define the following function to determine the optimal utility of a negotiation, where neither the client or supplier can increase their utility without a drop in their opponent's utility:

$$\text{OptU} = \sum_{i=1}^{q_n} \frac{\theta_i^c res_i^s + \theta_i^s res_i^c}{\theta_i^c + \theta_i^s}$$

4.3.1.2 Issues

The negotiations run in our experiments may use a number of issues. These issues are *independent* of each other — the value of one issue does not directly influence the value of another. This is a simplification of a real-world situation, but is necessary due to the complexity that dependent issues would introduce. For example, in a situation where a consumer wishes to subscribe to notifications about a particular topic for which the publisher charges a fee, the fee may be partially related to the frequency with which the notifications are requested. In our experiments, these issues are modelled as independent. All issues in the experiments are qualitative, and take numeric real values.

4.3.1.3 Tactics

The only type of tactics we are evaluating in this section is time-dependent tactics. We have used one of each of the three families of polynomial time-dependent tactic — Boulware ($\beta = 0.2$), linear ($\beta = 1$) and conceder ($\beta = 5$). The behaviour of these tactics can be seen in Figure 4.5. The experiments in this section use polynomial tactics, as their behaviour can be classed as simpler than exponential tactics (see Section 4.2.5.3 for details) We also ran the experiments with exponential tactics, and report on the difference in outcomes in Section 4.3.3.

Single tactics are used in each experiment, rather than using a strategy involving multiple tactics, because we are examining how negotiations perform with these tactics. Combining behaviours makes it more difficult to determine the cause of any changes in behaviour.

4.3.1.4 Time Model

In a real negotiation, the amount of time available in which to negotiate would be measured in terms of real time. In our simplified experiments, we use an interval-based time model, where a NC can send a single message in each time interval, which we refer to as a *tick*. In practical terms, this allows us to measure time by counting the number of messages sent, as in a full implementation, processing time would be negligible compared to transmission time. Increasing the deadline by one tick allows one extra message to be sent.

4.3.2 Hypotheses and Results

We evaluate DiNE using two criteria in this section: behaviour given different amounts of time in which to negotiate; and behaviour when the number of issues in the negotiation is varied. As the negotiations have been simplified to include only time-dependent tactics, deadline is the most significant factor of the evaluation since it alone controls the behaviour of the tactics.

4.3.2.1 Variable Deadline

In many situations requiring negotiation, a time limit is placed on a negotiation. This can be for one of many reasons. For example, a service could be attempting to set up subscriptions to an information source in order to monitor a particular task that starts at a predefined time. Everything needed to monitor the task must be in place before it begins, so any negotiation involved must have reached a mutually acceptable value within a specified time. Our hypothesis about the effect of time available in a negotiation is as follows:

If a negotiation has a large amount of time to complete, the outcome is closer to the optimal for both parties. In shorter negotiations, the difference between the utility for the client and that of the supplier is greater.

It may seem that with shorter deadlines, negotiations are more likely to fail completely and not find a mutually acceptable proposal. However, if both tactics are time-dependent and both offer their reservation values at the last possible chance, negotiations should *always* produce a successful outcome under the assumption that the environment in which they are negotiating allows for a mutually acceptable proposal to be found (i.e. there is some overlap in the acceptable regions of the preferences of both parties in the negotiation).

In order to test the hypothesis, negotiations were run through 500 environments using a single negotiation issue. The variable in this experiment is the amount of time available

in which to negotiate — the deadline, which was varied between 2 and 100 ticks. Negotiations take place in the same environments for each value of the deadline. For each combination of deadline and environment, the same negotiation was run with each type of tactic at each end.

Figure 4.7 shows that as the deadline increases, the utilities obtained by the client and the supplier get closer to each other, and also to the optimal utility. However, it does not show that increasing the deadline will lead to both parties receiving a *higher* utility. Instead, it appears that it is possible to get a better utility using a much shorter deadline. Closer examination of the graph reveals that when the client receives a higher than average utility, it is at the expense of the supplier, who receives a significantly worse utility. As the period of the oscillations in the graph is constant, it appears that this behaviour is predictable.

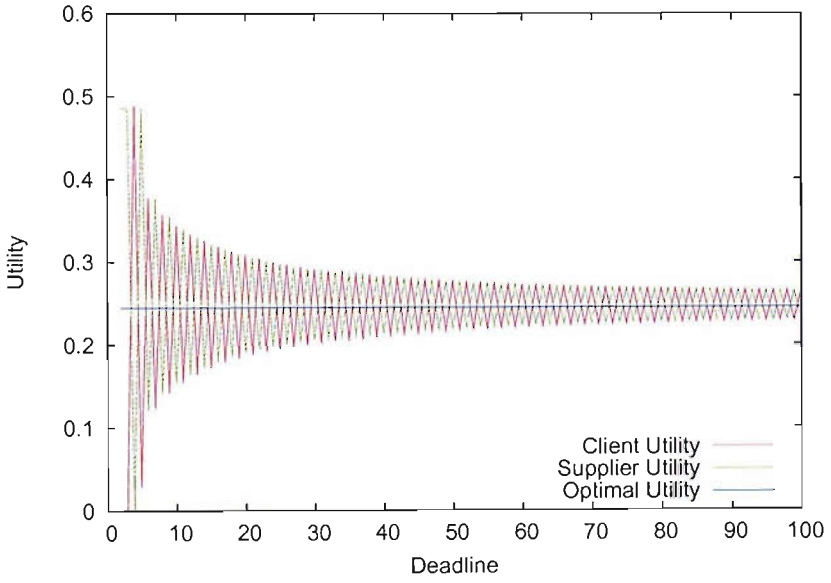

FIGURE 4.7: Average utility as deadline is varied

In Figure 4.8 we have divided the results into those with even values for the deadline and those with odd. From these graphs it can be seen that whenever an even value for utility is chosen, the client comes out significantly better off than the supplier. The situation is reversed using odd values for the deadline. The explanation for this behaviour is that if negotiations use all available time, for a given deadline, it will always be the same party offering their reservation value. Since the reservation value is the biggest concession a party is willing to make over the preferences for a particular negotiation issue or set of issues, it would normally be offered only when the alternative is for the negotiation to fail. Because the time model used in these experiments allows a single message to be composed and sent in each tick, the same party is always the one making the concession for a given deadline.

To illustrate the above behaviour, consider a simple negotiation between two parties,

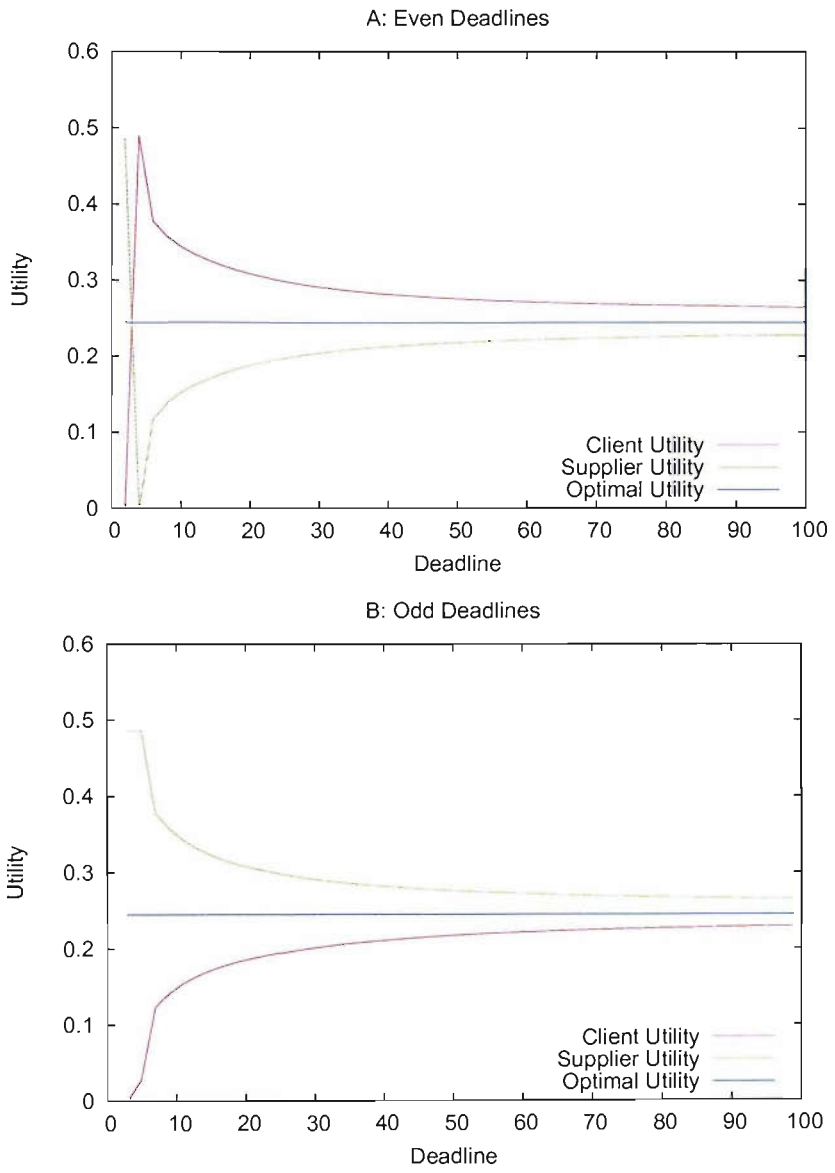

FIGURE 4.8: Average utility as deadline is varied through A) even values and B) odd values

in which there is time for four messages to be sent. The client makes the initial offer, and the supplier also responds with its initial offer. At this point, the client must make another offer. It knows that the supplier can still concede, so makes an offer normally. However, when the supplier comes to make the next offer it determines that this will be the last proposal offered, and therefore if the reservation value is not offered, the negotiation could fail.

Comparison of tactics

To further examine this situation, we split the results of the experiment even further in order to examine the behaviour of each type of time-dependent tactic in order to determine whether one type is more susceptible to this behaviour than another. These results are shown in Figure 4.9, in which the key part of the graphs is the shape of the

lines between each graph. This shows that over the course of many negotiations, a client gets the best utility from using a Boulware tactic, although this is at the expense of the supplier utility. Considering all parties involved, it is better for the client to use a linear time-dependent tactic, as this leads to the case where both the client and supplier utility cannot get much higher without sacrificing the other's utility, and is closer to the optimal utility.

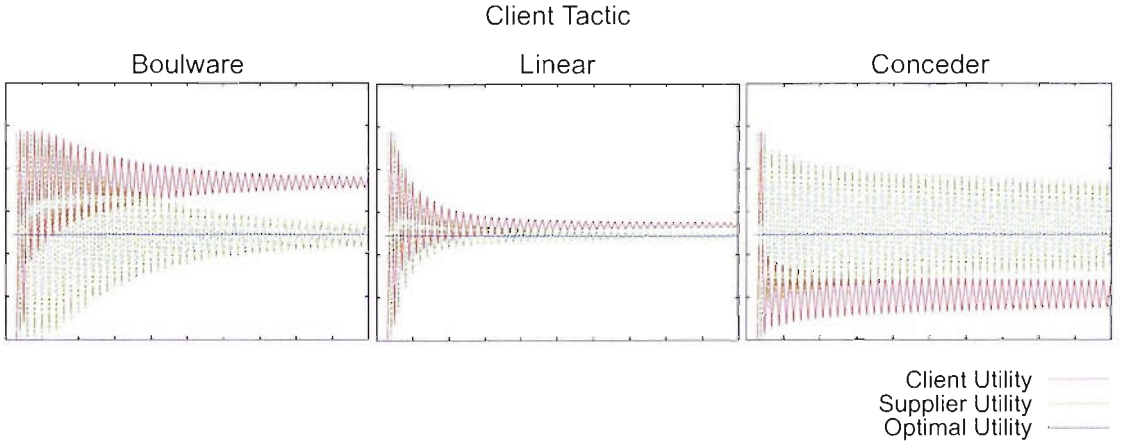

FIGURE 4.9: Average utility as deadline is varied with different client tactics (Client vs. Supplier)

In still more detail, the results are broken down into the individual tactic groups in Figure 4.10, which shows the results of each type of time-dependent tactic playing against each other. Above, we stated that it is better for the client to use a linear tactic. From this statement, it would be expected that linear versus linear tactics would produce a result closest to the optimal, and this is shown to be correct in the graph. The graphs also show that Boulware tactics often produce a better outcome than the other tactic groups, and conceder tactics can be expected to perform worse.

From these results, it would appear that this negotiation model is exploitable by one party to maximise their own utility at the expense of the others. The negotiation component initiating the negotiation could choose a deadline that is even and short, in order to force their opponent to offer the reservation value as soon as possible. A negotiation component's reservation value represents the lowest possible utility it could get from a negotiation if it succeeds. Additionally, the reservation value represents the highest possible utility the opponent could get from the negotiation. Forcing the opponent to offer their reservation value early would be the ideal way of winning a negotiation in a situation where each party is self-interested.

However, it must be noted that these experiments have been performed with many simplifications. It is the combination of these that have made this behaviour so apparent and easily defined. The main differences between these experiments and real-world versions are as follows:

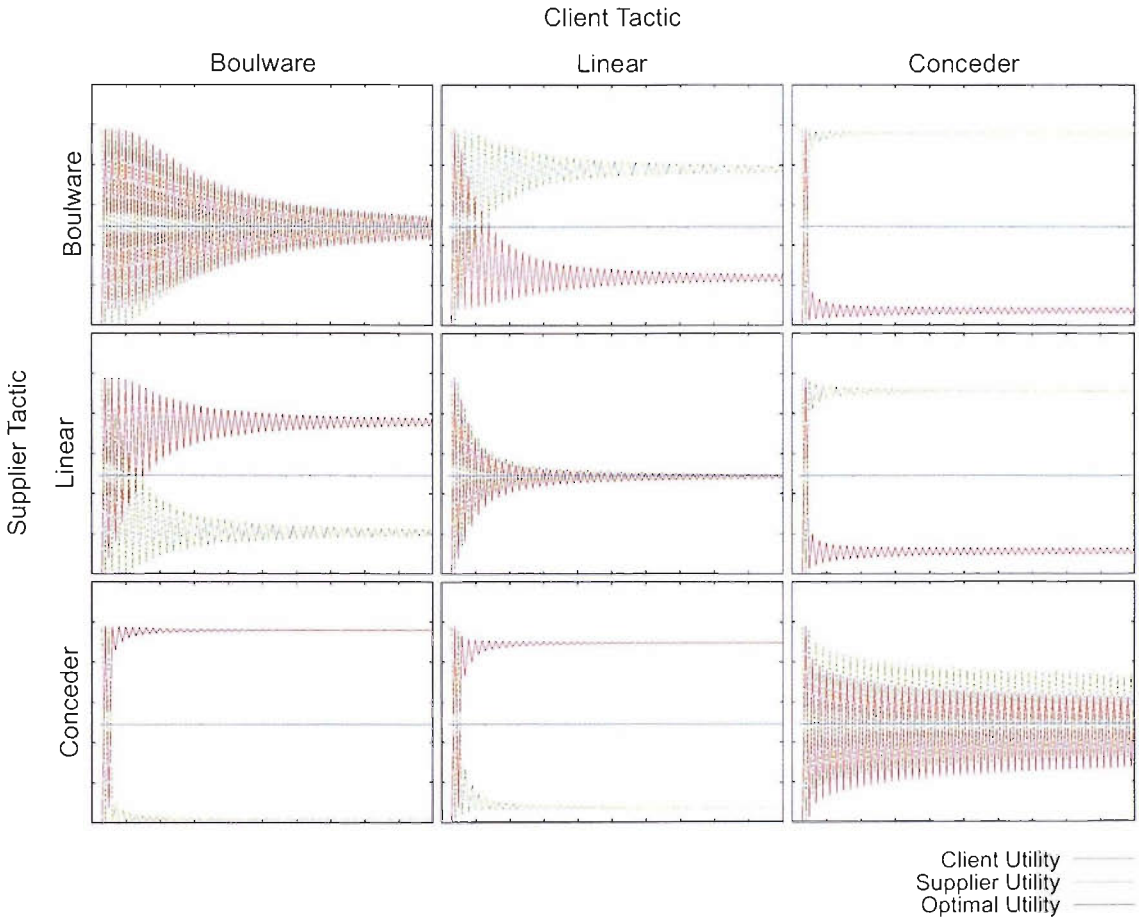

FIGURE 4.10: Average utilities as deadline is varied with different tactics (Client vs. Supplier

- *Interval-based time clock* — In these negotiations, the time taken to create a message or proposal is negligible, and a single message may be sent in every tick. In a real-world negotiation, an amount of time would elapse during the transmission of a proposal. If such a system used the Internet for communication, it is likely that this time would vary slightly between each message, especially for inter-site communication. This makes it difficult, if not impossible, to predict the number of messages that could be exchanged in a negotiation *before* it starts.

  Additionally, a single message is sent in every interval. The protocol does not specify that a negotiation component must reply to a message in the interval following the one in which it received the original proposal. This fits in with the real time aspect of a negotiation — if it is not possible to predict the amount of time taken to transmit a message, then it is impossible to determine when a reply must be received. Instead, a final deadline is imposed on the entire negotiation indicating the time by which a mutually acceptable proposal should be found.

- *Single tactics* — Only a single time-dependent tactic has been used in each one of these experiments. In negotiations for a service, one would expect a resource-

dependent tactic to be used in addition to the time-dependent tactic. This means that there are multiple tactics influencing the values chosen for each proposal, so the effect of the time-dependent tactic would be dampened by other additional tactics.

- *Tactic Behaviour* — A negotiation protocol specifies nothing about how a party should behave within the rules of the protocol, so it does not constrain the behaviour of each tactic. The tactics here all concede from their ideal value to their reservation value over the course of a negotiation. It may be desirable to prevent a tactic from conceding too quickly, resulting in the behaviour shown above. This could be done by having a minimum period before the concession value is offered, or using other methods. However, if the negotiation reaches the final opportunity to make a proposal and the reservation value is not offered, the negotiation may well fail. This may be an acceptable solution to a service provider that would rather trade off the possibility of losing business against being forced into offering a reservation value to a client.

These experiments show that increasing the amount of time available in which to negotiate does not necessarily increase the utility seen by either party. With increased deadlines, the utility of the two parties involved in a negotiation end up closer together than with a shorter deadline. With predictable message durations and fixed transmission times, it is possible to choose a deadline value that will maximise the utility of one party at the expense of another by forcing the other party to offer their reservation value early on. However, this behaviour would be harder to replicate in a real-world situation. This is investigated further later in this thesis, when a negotiation engine is integrated with the myGrid notification service. The experiments also show that a linear time-dependent tactic consistently produces a more predictable outcome, closer to the optimum for *both* parties than other time-dependent tactics.

4.3.2.2 Multiple Issues

The previous section concentrated on experiments using a single negotiation issue. Real world negotiations are likely to use multiple negotiation issues, as negotiation is often a trade-off of one issue against another. We expect that real world negotiations would use a small number of issues in order to facilitate trade-offs. With the behaviour using a single issue understood, we examine the implications of using multiple issues in a negotiation. Again, using only time-dependent tactics, the hypothesis about many issues is as follows:

As a negotiation involves more issues, the utility remains unchanged from that of negotiations involving a single issue (assuming all of the issues are independent). Taking the average length of a negotiation, increasing the number of issues should not make negotiations take any more time to complete, assuming that all issues concede independently.

In order to test this hypothesis, negotiations were run through 500 environments, in which the amount of time available for a negotiation was randomised. Using a random deadline removes the effect shown in the previous section where, for a given deadline, one party always has better utility than the other. The number of issues being negotiated over was varied between 1 and 25, and the average utilities are recorded along with the average number of messages exchanged in each negotiation.

As shown in Figure 4.11A, as the number of negotiation issues increases, the utility achieved by both the client and supplier does not change significantly. A few fluctuations are present, but these are due to the randomised nature of the environments in which negotiations are occuring.

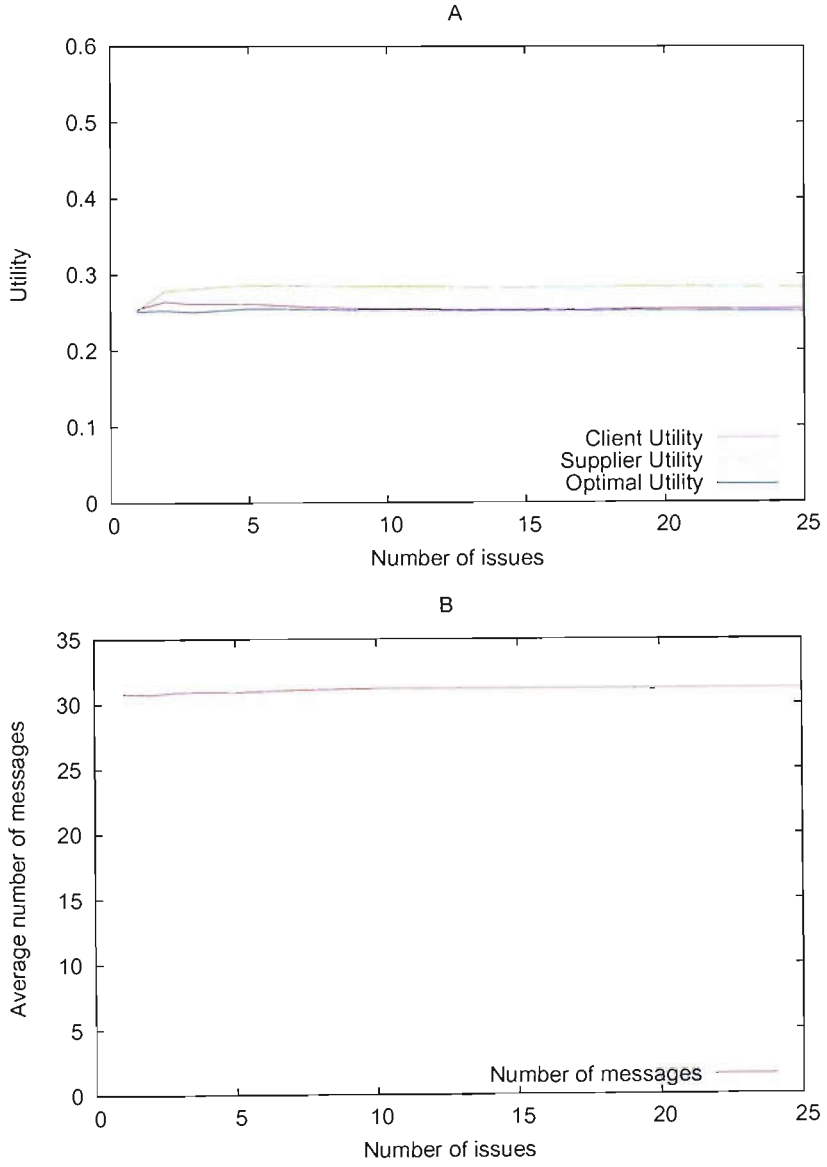

FIGURE 4.11: A) Average utility and B) Number of messages sent as number of issues varied

Figure 4.11B also shows that with a larger number of issues, the same number of mes-

sages are exchanged in reaching a mutually acceptable proposal. As the issues in these experiments are independent and concede as such, the amount of time the negotiation takes is constrained solely by the issue with the most restrictive preferences, regardless of how many other issues there are.

4.3.2.3 Execution Time

The previous experiments use an interval-based time model in which a single message can be sent each tick, allowing deadlines to be measured in terms of the number of messages that can be sent during the negotiation. This is based on the assumption that the time taken to transmit a message is the dominant factor in sending a message, and that the processing time is negligible in comparison. To validate this assumption, we measured the actual time taken to complete different negotiations. As the components of this test are directly coupled, there is no transmission delay and the measured time is purely processing time. Our processing time hypothesis is as follows:

The amount of time taken for a negotiation is directly proportional to the number of messages exchanged during the negotiation.

To confirm this hypothesis, negotiations in 500 different environments were run, each one being iterated 500 times to reduce inaccuracies in measurement. The amount of time taken for each was recorded, and collected to obtain the minimum, average and maximum amount of time taken against the number of messages sent in that negotiation.

Figure 4.12 shows that the mean time taken for negotiations varies linearly as the number of messages exchanged increases. The maximum time has a couple of fluctuations, which we have determined to be caused by garbage collection. These are rare enough not to affect the mean execution time. From the graph, we can conclude that the time taken is linearly related to the number of messages exchanged.

4.3.3 Experiments with Exponential Tactics

The experiments in the sections above were run with polynomial time-dependent tactics only. To verify that the behaviour observed from these experiments was not restricted to this class of time-dependent tactic, we repeated the experiments above using exponential time-dependent tactics instead of their polynomial counterparts. The behaviour of these tactics was shown earlier in Figure 4.5, and from this figure it can be seen that different values of the β parameter need to be chosen for these tactics. As such, we have used Boulware ($\beta = 1$), Linear[3] ($\beta = 5$). All other details of the experiments were kept the same.

[3] Exponential tactics do not exhibit a true linear behaviour, but we have selected a value for β that produces a curve closest to the linear behaviour of a polynomial tactic with $\beta = 1$.

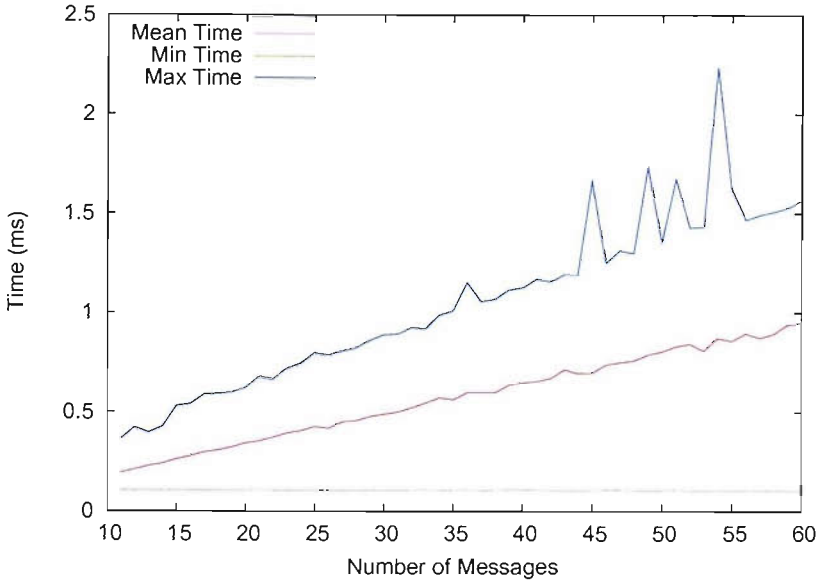

FIGURE 4.12: Graph of time taken for negotiations

In each experiment, the general pattern of the results was the same with exponential tactics as it is with polynomial tactics. Hence, we do not show the results here separately. Instead we say that the only differences observed were slight variations on the exact patterns of utility curves plotted from the results of the experiments. These results did not change any of the outcomes of the experiments. For comparison, we show the results of the variable deadline experiment side by side in Figure 4.13 below.

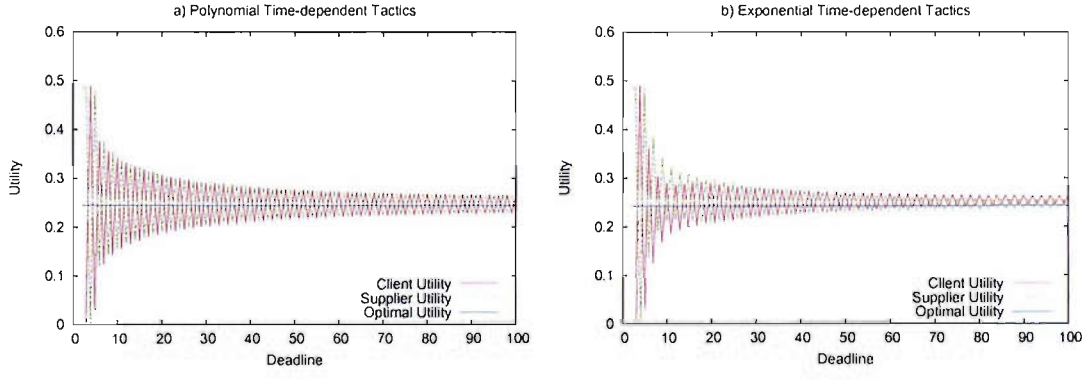

FIGURE 4.13: Average utility is deadline is varied with a) Polynomial time-dependent tactics and b) Exponential time-dependent tactics

4.4 Notification Service Scenario

In the previous section, we described experiments carried out using abstract issues and without a particular scenario in mind. In this section, we use a scenario in which the negotiation engine is integrated in the myGrid notification service.

To illustrate the functionality of Grid-based bioinformatics, myGrid has adopted an application that helps scientists study Graves' Disease, a hormonal disorder caused by over-stimulation of the thyrotrophin receptor by thyroid-stimulating antibodies secreted by lymphocytes of the immune system (Stevens et al., 2003). The Graves' Disease application follows an *in silico* experimental process typical of bioinformatics. In this process, the scientist attempts to discover information about candidate genes, makes an educated guess of the gene involved in the disease and then designs an experiment to be realised in the laboratory in order to validate the guess. This *in silico* experiment operates over the Grid, where resources are geographically distributed and managed by multiple institutions. It is a *data-intensive* Grid in which the complexity is in the data itself, the number of repositories and tools that need to be involved in the computations, and the heterogeneity of the data, operations and tools (Moreau et al., 2004). Users would like the Graves' disease experiment to be run repeatedly as new data is added to the knowledge base, and be notified of any changes in the results.

Our specific case experiment is a simplified version of this scenario — a notification service providing notifications from the SWISS-PROT database. SWISS-PROT is a curated protein sequence database providing a high level of annotation, minimal redundancy and high integration with other databases (EBI, 2003). It can be queried for sequences and annotations that are related to specified sequences, and is continually expanding: in the four months between two recent releases, the database grew by 7%, with an average of 890 changes per day[4].

For our example scenario, we assume 1000 consumers are interested in anything that matches 100 different protein sequences, and that a particular similarity search can be run with different precision. For simplicity, we represent precision by a number between 1 and 5, and we assume that a search with precision 2 takes twice as long as a search with precision 1. A particular consumer would like their search run as accurately as possible, every 8 hours. If we assume that a search with precision 1 takes 1 second, some 416 hours of CPU time are required every day (1000*100*(24/8)*5 seconds).

Negotiation is introduced into this experiment using two issues. Frequency represents the maximum number of hours between notifications: for the provider, the ideal value is 72 hours and the reservation value 12 hours; and for the consumer the ideal value is 8 hours and the reservation value 48 hours. The second issue is the number of iterations of the search. The provider prefers this to be between 1 and 5 and the client between 5 and 1. The preferences for the provider are kept constant, and a random variation is introduced into the client preferences to simulate different clients. Negotiation deadlines are between 30 and 60 messages.

We ran negotiations using the issues described above, and calculated the average outcome. The average frequency was 39.2 hours, and the average number of iterations

[4]http://www.expasy.org/sprot/relnotes/relstat.html

	CPU Time (sec)
With Negotiation	164,912
Without Negotiation	1,500,500

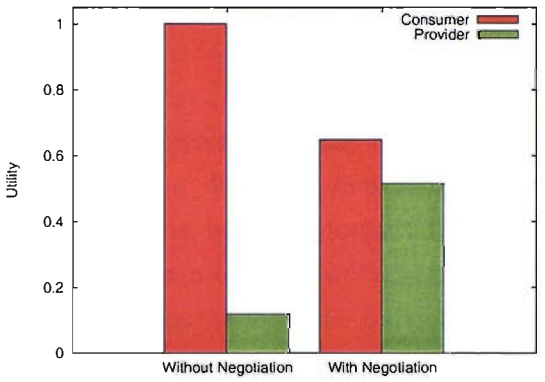

FIGURE 4.14: CPU Time and Utility with/without negotiation

was 2.69. These figures lead to 164,912 seconds of CPU time (45.8 hours), a reduction of 89% on simply allowing clients to request arbitrary Quality of Service levels. Figure 4.14 shows the difference in the amount of CPU time required when negotiation is introduced, and the corresponding differences in utility seen by both sides using the preferences above.

Introducing negotiation significantly lowers the amount of work the provider must do in this case, resulting in a significant increase in provider utility. However, this comes at the expense of the client utility; although there was a very large gap between the provider and client utility previously, the gap has now been reduced.

Introducing negotiation enables a provider to balance the utility of its clients with its workload. Although decreasing the client utility, lowering the amount of work required enables it to serve more clients while still satisfying them.

4.5 Summary

In this section, we have presented the design of DiNE, a Direct Negotiation Engine based on a version of NDF (Faratin, 2000), tailored for use in a situation such as a notification service. DiNE is a negotiation engine providing support to other services and applications wishing to support negotiation. DiNE allows the services using it to vary the level to which they are involved in the negotiation process, giving them access to influence the routines of proposal generation and evaluation. Proposals may be generated using a combination of system-supplied and host-supplied information, such as the level of available system resources.

We also presented an evaluation of the performance and behaviour of DiNE, using time-dependent tactics, a type of tactic which would be used in real-world applications of DiNE in conjunction with resource-dependent tactics taking system resources into account. This allowed us to observe the characteristics of the system using only these

tactics as different parameters are varied. Based on these experiments, we can make the following statements about DiNE:

- In Grid systems, time is often a critical resource. Agreements may have to be in place by a particular time in order for a service to run. Knowing how the amount of time available for negotiation affects the outcome is an important factor in choosing this deadline. In negotiations which have more time available in which to negotiate, results give utilities closer to the optimums for both parties. With shorter deadlines, the results are more unpredictable; the utilities of the client and supplier are further apart than with longer deadlines.

- Any negotiation mechanism should be fair to all parties involved. If one party can predict the outcome of the negotiation before it is run, it can be unfair to the other party. If the message transmission time is predictable, it is possible for the client to initiate a negotiation using a deadline chosen to produce significantly better results for the client at the expense of the supplier. This is due to the client being able to predict that the supplier will have to offer its reservation value at the expiry of the deadline, causing the supplier to receive the lowest possible utility. Unpredictable message transmission times in a real-world implementation are expected to mask this effect, making it harder to select a deadline chosen to cause this behaviour.

- Negotiations are typically undertaken over multiple issues, trading one issue off against another one. Negotiations involving multiple issues do not take any longer to complete than negotiations with a single issue, assuming that each issue is independent and concedes as such.

- When choosing a deadline for negotiation, it is useful to know that there will be no factors other than the deadline affecting the amount of time taken. As negotiations take longer in terms of the number of messages sent, the processing time taken to perform the negotiation is linearly related to the number of messages — the system scales predictably.

DiNE was designed with the context of a notification service in mind, where consumers and publishers of information have conflicting preferences about delivery conditions for a subscription to notifications. To show the intended benefit of such a configuration, our experiment simulating this scenario showed that negotiation enables the load on the service provider to be reduced, and the number of clients able to be served to be increased.

In summary, DiNE enables automated negotiation to occur between two services, allowing one to request a service from another while specifying conditions about the quality of service with which the service is delivered, and having differences in the preferences

about these conditions automatically resolved by DiNE. The evaluation of this negotiation is novel work, as it examines the behaviour of time-dependent tactics in more depth than other work, discovering the precise behaviour of the tactics with respect to predictable deadlines.

The contributions presented in this chapter were an empirical evaluation of a heuristic-based bilateral negotiation model, showing the behaviour of the model in an isolated environment; and a demonstration that negotiation is a suitable model for resolving differences between the QoS preferences of a consumer and a service provider in a distributed system.

DiNE is a direct, bilateral negotiation engine. To enable negotiation in a distributed notification service, we need an extension of DiNE where intermediaries can contribute to a negotiation, redistributing or reselling items obtained for previous clients. In the next chapter, we present a model of *chained negotiation* enabling this, and develop *ChANE*, a chained negotiation engine.

Chapter 5

Chained Negotiation Engine

In Chapter 4, we established that direct negotiation provides one way of resolving the differences in requirements between consumers and publishers of notifications in a notification service. We developed a negotiation engine, DiNE, to enable mutually acceptable values for notification parameters to be automatically determined for directly connected producers and consumers, and demonstrated that, by using the negotiation engine to manage client demands, the load on the service provider can be reduced, enabling more clients to use the same service. In this chapter, we present an evolution of DiNE, ChaNE (Chained Negotiation Engine), which supports negotiation via intermediaries (or middlemen).

5.1 Introduction

In large-scale deployments of notification services, multiple instances of a NS are likely to be hosted at different locations (Krishna et al., 2003) and, consequently, publishers and consumers may interact with different NS instances. Typically, they publish messages to, or consume them from, their local NS. Such distribution of NSs offers better scalability (by avoiding a single NS being responsible for all notifications) and better security (by allowing NSs dealing with private topics to be hosted behind firewalls). It is natural, therefore, to expect such distributed NSs to be networked (similar to news servers) and to be capable of propagating notifications between publishers and consumers. In this view, as a result, a common configuration pattern consists of several NSs chained between a consumer and publisher; hence, publishers and consumers may be separated by several NSs, which we also refer to as *intermediaries* or *middlemen*.

Although distributed notification is more complex, it offers some potential efficiency gains. Instead of sending the same notification messages to multiple consumers subscribed to the same topic, it becomes possible to propagate a single instance of such

a message between NSs to reduce network traffic. We refer to such an optimisation as *sharing notifications*. The difficulty with this, however, is that if consumers can negotiate QoS parameters for a subscription, two sets of requirements may be sufficiently different that they preclude the sharing of notifications, and hence may impose a higher load on the network of NSs.

Our direct negotiation engine (DiNE) discussed previously in Chapter 4 is unsuitable in this context, as it was aimed at directly connected consumers and producers. With distributed notification services, by contrast, consumers and publishers are connected through a series of middlemen, which can in turn have their own QoS requirements. In response, we have designed a negotiation model, which we refer to as *chained negotiation*, in which the consumer and publisher no longer communicate directly; instead, negotiation takes place through middlemen, which pass service proposals back and forth between publishers and consumers, potentially modifying them in order to satisfy their own QoS requirements.

Furthermore, chained negotiation is designed to promote sharing of notifications. Specifically, middlemen record previous commitments of publishers to provide QoS at given levels; they not only pass proposals but also attempt to identify existing commitments that can be reused to satisfy consumer requirements, potentially impacting on the negotiation outcome for the consumer or the publisher. Using these existing commitments to satisfy a new consumer represents a gain in efficiency, as without it, a new commitment would need to be made between the consumer and the publisher.

In this chapter, we discuss the design and evaluation of ChaNE, a negotiation engine supporting chained negotiation. To demonstrate its effectiveness, we have designed a series of experiments that aim to show that using a chained negotiation system in conjunction with a distributed notification service enables more consumers to be served by a given set of notification services, and that this can be done at the same time as reducing the load on service providers.

This chapter provides a novel contribution — chained negotiation is a new form of negotiation that deals with the situation where negotiation takes place through intermediaries, enabling the redistribution and/or reselling of items obtained through negotiation, and allowing the sharing of subscriptions between intermediaries, reducing the overall load on the system by enabling notifications to be shared between intermediaries and consumers. Our empirical evaluation demonstrates that chained negotiation is an effective way of enabling a consumer and a service provider to resolve differences in their preferences over quality of service, while providing a mechanism for an intermediary to supply the requested item or service if it already has a commitment to provide the item to another consumer.

The rest of this chapter is organised as follows. In Section 5.2 we present chained negotiation, and the design of ChaNE, our implementation of chained negotiation. In

Section 5.3 we evaluate the behaviour of ChaNE, to determine the costs and benefits associated with chained negotiation. Section 5.4 describes a simulation of the intended use of chained negotiation, a distributed notification service. Finally, we summarise the chapter in Section 5.5.

5.2 Design of a Chained Negotiation Engine

In this section we present ChaNE, our Chained Negotiation Engine. ChaNE is an evolution of DiNE, and as such builds on many concepts introduced in Section 4.2.2. ChaNE encapsulates the behaviour of chained negotiation, enabling other systems to make use of chained negotiation functionality.

5.2.1 Core Concepts

Before discussing the design of ChaNE, we introduce some core concepts which will be used during discussion of chained negotiation, and ChaNE in particular. Firstly, we recall that direct negotiation takes place between one or more *clients* attempting to obtain products or services from one or more *suppliers*. Both parties exchange proposals to find a mutually acceptable price or set of constraints for the negotiation item. By contrast, *chained negotiation* is an extended form of negotiation, in which a client initiates a negotiation with a *middleman*. If the middleman cannot provide the requested item itself, it will in turn initiate negotiation with a supplier, or potentially another middleman. Thus a chain is formed between the client and the supplier.[1] More specifically, middlemen forward proposals between the clients and suppliers so that an agreement can be reached, as illustrated in Figure 5.1, where the *Client* is negotiating with the *Supplier* via *Middleman*1 and *Middleman*2. The client and supplier at each end of the negotiation chain are referred to as *end NCs*. Messages sent towards the supplier are referred to as *upstream*, whereas messages sent towards the client are referred to as *downstream* messages. Middleman also have ability to modify the messages that they send between the client and supplier. For example, a commercial service redistributing notifications on a particular topic may have to pay for the initial subscriptions. To recoup this outlay, and to provide a return for offering the service, it may add an amount that it will retain to the price paid by the consumer.

Chained negotiation is the general term for negotiations involving middlemen. When a chained negotiation terminates successfully, it can be placed into one of two classes:

[1] It is not our intention that chained negotiation deals with the selection of negotiation partners. In DiNE, we leave the selection of supplier to an external mechanism such as a directory service. Similarly, for chained negotiation, we assume that an external process creates the chain between clients, middlemen and suppliers.

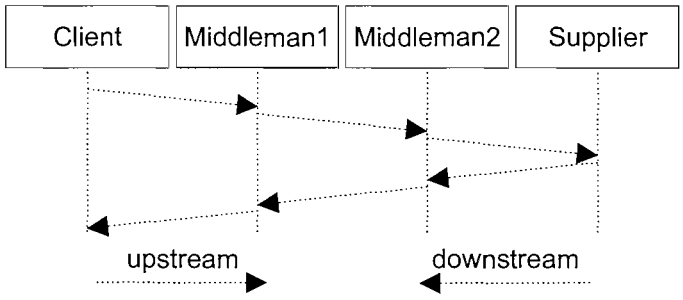

FIGURE 5.1: Message exchange sequence in chained negotiation

matched negotiation, which satisfies the client's request by using an existing commitment; and *unmatched* negotiation, which requires a new commitment to be made upstream. We also define *forwarded* negotiation as a variant of chained negotiation where no existing commitments are used. In the context of a notification service, existing commitments are represented by subscriptions held for previous consumers.

5.2.2 Negotiation Protocol

The negotiation protocol defines the rules of the negotiation — the parts of the negotiation that must remain constant no matter how a participant behaves. Specifying this allows a clear distinction to be drawn between the actions that are taken because they are specified by the protocol, and those which have been taken due to the influence of a negotiation strategy. The protocol consists of the valid participant types, the interactions between those participants, the format of the messages, and rules that must be followed.

5.2.2.1 Participant Types

In a chained negotiation, there are three types of participant: client, middleman and supplier. A client is interested in obtaining a product or a service from a supplier, and typically has a set of preferences for the ranges of values considered acceptable for each issue in a negotiation. A supplier is attempting to provide products or services to interested parties. A middleman can act as both a client by obtaining an item from another supplier, and a supplier by reselling or redistributing goods or services to other parties.

5.2.2.2 Interactions in Chained Negotiation

The interactions in chained negotiation are similar to those in direct negotiation, with some additional messages to enable multiple parties to agree to a commitment. These

interactions are shown in Figure 5.2, and described below. Interactions that differ from those in direct negotiation are marked with a (*) in the text.

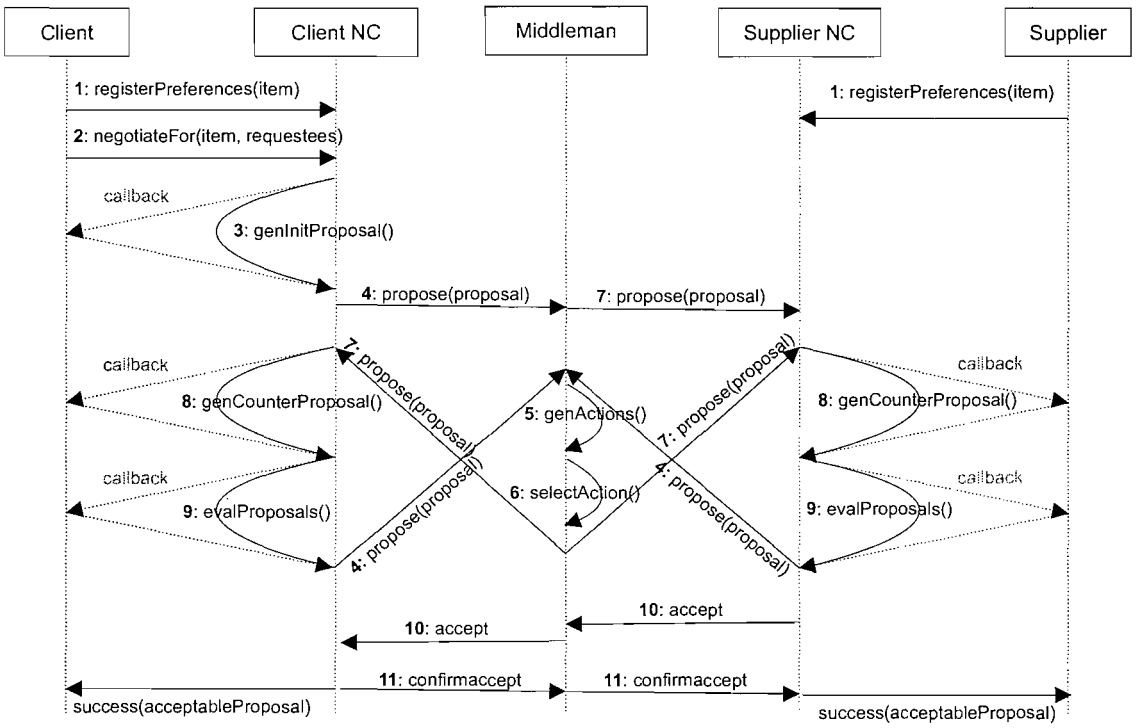

FIGURE 5.2: Interactions in a chained negotiation

1. Preference Registration

   Before the negotiation begins, both parties initialise their respective negotiation component (NC) with a set of preferences for the items they are interested in or are making available. The preferences consist of two values, an ideal value and a reservation value which, combined, represent the extent of the range the host considers acceptable for a specific issue.

2. Negotiation Initiation

   The client gives its NC an instruction to begin the negotiation process. In order to constrain the amount of time a negotiation can take, a deadline is specified as the negotiation is started. Not only does this ensure that a negotiation will complete, it also ensures that if the item is required in a specific time frame, the negotiation completes in time for delivery to take place.

3. Initial Proposal Generation

   The client's NC generates an initial proposal ready to send to the supplier's negotiation component. Typically, this would be the client's ideal values for all of the issues involved in the negotiation. Optionally, this step can involve callbacks to the client in order that the client may influence the initial proposal generation.

4. Proposal Transmission

   The negotiation process now enters a cycle, initiated by a proposal being transmitted. The communication mechanism is left unspecified — the host must implement a reliable message transport for message delivery, as no checks are performed to ensure message delivery. The message is sent to the closest middleman.

5. Action Generation (*)

   Once a middleman receives a proposal, it must decide what to do with it. The protocol does not specify behaviour here; only that it should transmit a proposal to either the upstream or the downstream party in the negotiation. The proposal can be forwarded unaltered, or the middleman could change some values for the issues in the proposal. A series of actions are generated covering the different messages to consider sending.

6. Action Selection (*)

   The list of actions generated in the previous stage are evaluated, and the best one is selected for execution. The message from this action is sent to the intended destination in the next stage.

7. Proposal Transmission/Forwarding (*)

   The message from the action selected in the previous stage is sent onto the next NC in the chain — this could be an end NC, or another middleman. Eventually, this stage results in a message reaching an end NC.

8. Counter-proposal generation

   When a proposal is received by an end NC, a counter-proposal is generated. Further details of proposal generation by end NCs can be found in Section 4.2.5.3. As with the initial proposal generation, counter-proposal generation can involve callbacks to the NC in order to retrieve external environmental conditions such as the current system load, or to allow p to provide a custom proposal generation routine.

9. Proposal Evaluation

   Once the counter-proposal is generated, both the generated counter-proposal and the recently received proposal are evaluated using p's utility functions. A comparison is made between the score of the counter-proposal and the received proposal. If the counter-proposal has a higher utility, it is transmitted to the other negotiation component (return to step 4). If the received proposal has equal or higher utility than the generated counter-proposal, the situation is considered acceptable.

10. Proposal Acceptance (*)

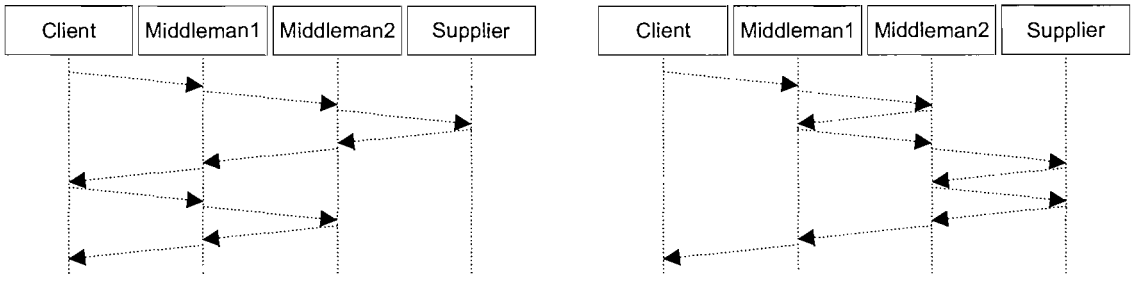

FIGURE 5.3: Example of Interactions in a chained negotiation

Once an acceptable state has been reached, one party issues an accept message to indicate that they consider the values for the issues in the proposal to be acceptable. This does not cause a commitment to be made; it merely signifies that the party is *prepared* to commit to the values in the acceptable proposal. A reliable communication mechanism is assumed, as each party needs to be sure that all involved parties are prepared to commit before proceeding.

11. Make Commitment (*)

    Once an accept message reaches the client in the negotiation, a state has been reached where there exists a party somewhere upstream of the client that is prepared to supply the client with the requested negotiation item using the values for the issues in the acceptable proposal. At this point the client sends a message upstream, initiating a commitment. This message is propagated further upstream by the middlemen to the party supplying the item (this could either be the supplier or one of the middlemen in the chain).

For the client and supplier, the interactions they have when negotiating directly with each other are identical to those when interacting with a middleman, therefore there is no difference for them between direct and chained negotiation in ChaNE.

The different combinations of these interactions form *interaction patterns*. Interaction patterns are affected by a number of factors, including the number of middleman in a negotiation, the number of existing commitments held by the middlemen, and the strategies and preferences in place at the middlemen. For this reason it is not possible to show all possible interaction patterns, but examples *permitted by the protocol* are given in Figure 5.3. It is worth noting that although these interactions are possible within the protocol, the figure on the right requires the strategy for a middleman be capable of generating proposals not based on simply forwarding or offering existing commitments — the strategy which we will present does not support this and it is left for future work.

The pattern of using Accept and ConfirmAccept messages is used in response to the difficulty of enabling acceptance by multiple parties. When only two parties are involved, the presence of an Accept message indicates that the other party accepts the proposal,

and this can be turned into a commitment very easily. When more than two parties are involved, all have to "agree to agree" on something first — Accept messages indicate that each party agrees to a particular proposal, and then a ConfirmAccept message is used to initiate turning the agreement into a commitment. This is a variation of the two-phase commit approach (Gray, 1978) used for distributed databases to commit an action atomically. In DiNE, commitments are not actually made within the NCs — it is left to the host components to arrange a commitment separately. We chose to add commitment making into ChaNE rather than have hosts arrange this because of the added complexity of getting multiple parties prepared to commit to something, and because otherwise, clients and suppliers would need to be aware that they are part of a chained negotiation, interacting with all of the involved parties.

5.2.2.3 Message Structure

In the process of a chained negotiation, messages are exchanged between parties conveying offers and other information. Four different types of message are used:

- Propose

  Propose messages convey an offer from one NC to another, with a set of values proposed by the sender for each of the issues in the negotiation.

- Accept

  Accept messages indicate that the sender is prepared to accept the proposal referenced in the message.

- ConfirmAccept

  A ConfirmAccept message indicates that a commitment is being made to the proposal referenced in the message. No party should receive a ConfirmAccept message referencing a proposal they have not previously accepted.

- Terminate

  Terminate messages are used to signify a negotiation failure or explicit termination. A reason field is contained in the message to indicate to the other party why the negotiation failed.

As in the messages used in direct negotiation, all of the messages contain a common message header, used for identifying the type of message, sender and recipient, and the negotiation it is related to. Additionally, chained negotiation messages also contain fields holding the distance upstream and downstream, measured in number of hops, to each end of the negotiation chain. The need for this field is explained in Section 5.2.2.5. These fields are populated by the end NC setting the distance from itself to zero when a

Message	Field Name	Description
(All)	MessageType	Identifies the type of this negotiation message
	SenderID	Identifier for the sender of this message
	RecipientID	Identifier for the recipient of this message
	NegotiationID	Unique identifier for this negotiation
	DistUpstream	Distance between sender and supplier in hops
	DistDownstream	Distance between sender and client in hops
Propose	ProposalID	Unique identifier for proposal in the negotiation
	Item	Item this negotiation is for
	Elements	List of values for issues in this negotiation
	TimeRemaining	Amount of time remaining in this negotiation
Accept	ProposalID	ID of the proposal being accepted
	TimeRemaining	Amount of time remaining in this negotiation
ConfirmAccept	ProposalID	ID of proposal that was accepted
	TimeRemaining	Amount of time remaining in this negotiation
Terminate	Reason	Reason for the negotiation being terminated

TABLE 5.1: Chained Negotiation Message Structure

message is sent, and by middlemen incrementing the field for the distance in which they are sending. Details of the fields found in the message header, as well as the specific messages, may be found in Table 5.1.

5.2.2.4 Rules

Chained negotiation is governed by a set of rules ensuring that all negotiations follow a prescribed pattern and have sufficient information to complete successfully within a specified period, assuming that the preferences of the parties involved overlap sufficiently so that an agreement is possible. If the preferences do not overlap, no agreement can be reached. These rules are discussed below.

1. NCs upstream of the client can only send Accept messages downstream if they can provide the required item.

   For any negotiation component further upstream than the client (i.e. the supplier or any of the middlemen), they must not send an accept message downstream without first having either:

   (a) matched the current negotiation to an existing commitment it holds that is capable of fulfilling the negotiation; or

   (b) received an accept message from the party directly upstream in the negotiation, indicating that they are definitely able to supply the item under the conditions in the accepted proposal.

   If this rule is adhered to, the case where a client agrees to buy something the seller cannot actually provide will not arise. The rule is not enforcible — the protocol

does not allow the client to verify that there is an upstream agreement in place to satisfy an accepted proposal. However, it is *rational* that this rule would be obeyed — failure to deliver an item for which a contract has been made could result in penalties or loss of reputation (Zacharia et al., 1999). Verifying compliance with this rule is beyond the scope of this work and is left for future work.

2. Propose and Accept messages can only be sent if there is time remaining before the deadline.

   The deadline specified by a negotiation indicates the amount of time remaining in which an acceptable proposal can be found. By the time the deadline expires, the client should have received an Accept message from upstream. If this is the case, Rule 1 means that all parties involved in the negotiation have found an acceptable proposal and are prepared to make a commitment. All that remains is for the client to initiate the ConfirmAccept messages. This message can be sent at any time up to and including the deadline, as it will be propagated to the party fulfilling the negotiation without delay.

3. Only send a message if the reply can reach the client before the deadline.

   Proposals are often generated using the amount of time remaining before the deadline as one of the controlling factors in the rate of concession (see Section 4.2.5.3). The amount of time remaining before the deadline also influences which direction a middleman should choose to negotiate in. Rule 2 states that the negotiation must have found an acceptable proposal by the time the deadline expires if the negotiation is to be successful. For this to be possible, a middleman must ensure that before sending a message to an upstream party there is enough time remaining for the reply to be received and forwarded back to the client before the deadline expires. Implications of this rule are discussed below.

5.2.2.5 Distance to Client and Supplier

In order to satisfy Rule 3 above, it must be possible for a middleman to determine how long it will take for a reply to reach the client. This is done by adding a field to the messages to hold the distance to the client, measured in number of hops. It is also possible to hold the distance to the supplier using the same method. This section discusses the implications of using these distances.

Without knowing the distance to client and supplier a party is negotiating with, they must make an assumption about the distance. Clients and suppliers know their downstream and upstream distances respectively are both 0, but may not know how far away the other is. Middlemen can be an unknown distance from both ends. A middleman could assume that it is directly downstream and upstream of the party, i.e. a distance of 1 in each direction, but this assumption being incorrect causes two separate but related

problems: an inability to determine whether there is time to negotiate upstream; and an inability to correctly determine the amount of time remaining before a final concession can be made.

If a NC is unable to correctly detect that there is sufficient time to send a proposal upstream so that a reply can reach the client, it will not be possible for an Accept message to reach the client before the deadline expires. Rule 2 states that only ConfirmAccept messages can be sent after the deadline has expired, and Rule 1 means that only the client can initiate ConfirmAccept messages. Because the client will not receive an Accept message, the negotiation will fail due to deadline expiry. This is illustrated in Figure 5.4, where a chained negotiation is taking place with two middlemen. *Middleman*2 does not know the distance to the client, and assumes this to be 1. This enables it to send a proposal upstream to the supplier, although there is no time for the reply to reach the client, so the negotiation fails due to deadline expiry at *Middleman*1.

The second problem arising from not knowing the distance to the client and supplier is that it is impossible to accurately determine the last possible moment at which a final concession should be made. With time-dependent tactics (see Section 4.2.5.3), the reservation value is offered at the last possible moment. The last possible moment is not at the deadline, as it would not be possible for this final offer to be received by anyone. Instead, it must be made at the latest time at which it is possible for the message to be received by the opponent, and for a reply to reach the client before the deadline expires.

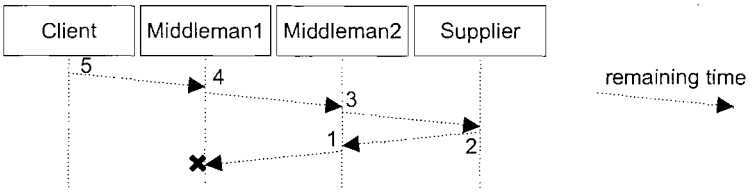

FIGURE 5.4: Example of negotiation failure when not using distance to client

To illustrate this problem, consider the example shown in Figure 5.5. A client c and supplier s are negotiating for an item via middleman m_1, with the negotiation involving a single issue. The preferences of both parties have an overlapping region, so that it is possible for an agreement to be made. The acceptable range of c for the preference is [10:50], and for s is [80:40]. Proposals are forwarded unchanged by m_1. The negotiation begins with an initial deadline of 8. In this example, s knows the distance to c is 2, but c does not know the distance to s and assumes it is 1.

The first proposals issued by c and s contain their ideal values from their preferences, indicating what value they would like to get. This causes a problem when c receives the proposal from s, since the value in the proposal is not acceptable and a counter-proposal is generated. The remaining time at this point is 4, indicating that there is still enough time to make a proposal to the party directly upstream, receive a reply, and then offer the reservation value if necessary in the next step. However, the party

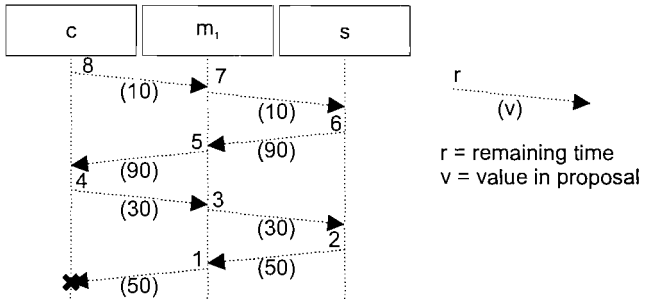

FIGURE 5.5: Example of reservation value being missed not using client distance

directly upstream is actually a middleman forwarding the proposals onto s. This causes the message to take longer than expected to come back. Supplier s correctly realises that as the remaining time is now 2, it is its last chance to make an offer in time to get to c, so offers the reservation value. This value is actually within the preferences of c, but as the deadline has elapsed and c has not received an accept message, it is impossible to make a commitment. If c had known that the distance to s was actually 2, when the remaining time was 4 it would have offered its reservation value, which would have been accepted by s and the negotiation would have been successful.

To avoid the problems discussed in this section, all messages sent in chained negotiation carry a field containing the distance of the message sender to both the client and supplier. The contents of these fields are calculated automatically — the client and supplier set the distance to themselves to be zero when they send a message, and when middlemen send a message upstream, they increment the distance downstream by one (and vice versa). The client and middlemen do not know the distance to the supplier until a message has been received by the supplier and a reply sent downstream, enabling the distances to be calculated by all of the NCs in the chain.

5.2.3 Negotiation Strategies

The previous section described the negotiation protocol — the rules that constrain all behaviour within chained negotiation. However, actions within this protocol have to be defined before chained negotiation can occur. The negotiation strategy controls how each participant behaves within the protocol. In chained negotiation, the behaviour of the client and supplier is no different from parties in direct negotiation, albeit with the minor modifications to the protocol described in the previous section. For this reason, the strategy described below applies only to middlemen, as the reasoning involved in middlemen is significantly different to clients and suppliers.

Figure 5.6 shows an overview of the data flow in the negotiation strategy. A context store holds information on every negotiation, including all messages sent and received so far, and details of the participants and item in the negotiation. The commitment store is used to record details of commitments that have already been made for previous clients.

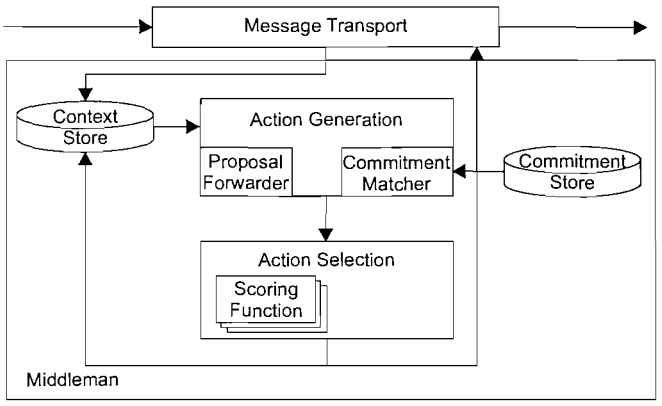

FIGURE 5.6: Data flow in ChaNE

Messages are received using the Message Transport layer and stored in the context store, before the action generation routine uses the context and commitment stores to generate a series of potential actions to be taken. Actions are messages that can potentially be sent. The best action is then selected using the action selection routine, and the message from that action is added to the context store before being sent to its recipient using the message transport. The action generation and selection routines are described below.

5.2.3.1 Action Generation

Actions are the major part of chained negotiation — they comprise a message which would be sent if that action was to be chosen. This message can be either a proposal, or an accept message. Actions are generated in two main ways:

- Commitment Matching: Proximity functions (described below) are used to select existing commitments that are close (i.e. similar values for the issues under negotiation) to the last received proposal from downstream. As they are close to the last received proposal, they are more likely to be acceptable to that party than one further away. The commitments selected by the proximity functions are made into proposals to be sent back downstream. No upstream proposals are generated using this method.

- Proposal Forwarding: The last received proposal from upstream is copied as a proposal to be sent downstream, and vice versa. The proposal elements are left unchanged.

- Proposal Modification: The last received proposal from upstream is copied as a proposal to be sent downstream, and vice versa. During the process of copying the proposal elements, some of them may be altered by the middleman, for example so that it can take some of the cost element of the proposal as profit for operating the service.

Not all of the above methods are used to generate actions in all negotiations — negotiations where the middleman is not trying to make a profit do not use the proposal modification method, for example. Each method may generate any number of actions, which are evaluated using the scoring functions described below. It is also possible to add extra routines for generating actions here — potential future work includes generating proposals speculatively, allowing a middleman to anticipate high demand for an item and try to obtain this at a high QoS to potentially satisfy many clients.

5.2.3.2 Proximity Functions

Chained negotiation uses *proximity* functions to determine whether one proposal is *satisfiable* by another proposal or commitment. Proposal p_1 is satisfiable by p_2 if each element of p_2 is at least as good as its counterpart in p_1 (from the point of view of the sender of p_1), and the negotiation item of each proposal is the same.

$$prox(p_1, p_2) = \begin{cases} -1 & \text{if Subj}(p_1) \neq \text{Subj}(p_2) \\ 0 & \text{if Elements}(p_1) = \text{Elements}(p_2) \\ < 0 & \text{if } p_1 \text{ not satisfiable by } p_2 \\ \geq 0 & \text{if } p_1 \text{ satisfiable by } p_2 \end{cases}$$

Proximity functions are used to determine appropriate existing commitments. A threshold is used to determine which commitments are appropriate — if the middleman is only forwarding proposals, a threshold of 1 will ensure that any time a commitment is selected, it should be accepted by the client. A threshold of slightly less than 1 allows commitments that are very close to be selected, and these might be acceptable when the client makes its next concession. However, in the next section we will show that proximity functions are also used by scoring functions to rank actions, and using a threshold of less than 1 can cause a problem here when matched commitments that are very close to, but less than, 1 are ranked higher than a forwarded proposal close to the deadline, causing the negotiation to potentially fail if it is not accepted.

5.2.3.3 Action Selection

Each time a message is received, the best course of action must be selected. To determine this, chained negotiation uses *scoring functions*, which assign a score to a particular action. They are required because multiple actions are generated every time, and the most appropriate one must be selected. ChaNE has been designed to allow additional scoring functions to be supplied by the host, but core scoring functions are necessary to control the behaviour of chained negotiation. We have defined the following core scoring functions:

- Proximity

  Proximity functions as described in Section 5.2.3.2 are used to select actions that are closer to being acceptable, which occurs when the proposal they contain is closer to the last received proposal from the party to which the action will be sent. This allows different ways of generating proposals to be used, and the best one selected. Actions containing proposals that are close to the last received proposal are given a higher score than those that are further away by this scoring function.

- Acceptable actions

  This function gives a higher score for actions that would directly lead to an acceptable state, such as accepting a proposal already received. This function additionally gives a higher score for an action that would reuse an existing commitment. It favours actions that complete the negotiation earlier, and those that incur less cost by reusing an existing commitment.

- Path least recently chosen

  In the early stages of a negotiation, there may be no reason to score the forwarding of a proposal upstream any differently to sending a proposal downstream. This could lead to the situation where the negotiation does not proceed because no messages are ever forwarded upstream. Considering the two directions in which to negotiate, this function gives a higher score for the direction least recently used. For example, if the last message was sent upstream, downstream actions get a higher score. If the last five messages were sent upstream, downstream actions get a significantly higher score from this function. The scoring function can also be used to bias a middleman into favouring one side of a negotiation. For example, it may be configured to do as much of the negotiation as possible without involving the client, which could be on a low-power device or connected via an unreliable or slow connection.

Actions are evaluated using a weighted average of all of the configured scoring functions.

5.2.3.4 Negotiation Algorithm

The significant part of chained negotiation occurs when a message is received by a middleman, which generates a set of possible *actions* (each including a message to be sent), and then executes the best action, as shown in Algorithm 3.

Initially, the most recently received message from each side is collected. The middleman looks for a commitment that can satisfy the last message from downstream using a proximity function, as described above. If one is found, or if an accept message from upstream has been received matching the downstream message, an action is generated

Algorithm 3 Processing Propose and Accept messages

```
msg_d =getLastDownstreamMessage()
msg_u =getLastUpstreamMessage()
msg =mostRecentMessage(msg_d, msg_u)

if remainingTime = 0 then terminateWithFailure()
if findCommit(msg_d) or hasUpstreamAccept(msg_d) then
  actions.add(makeAction(ACCEPT, down, msg_d))
end if
if t_rem ≥ (dist_D + 2 * dist_U) then
  actions.add(makeAction(PROPOSE, up, msg_d))
end if
if msg_u != null then
  actions.add(makeAction(PROPOSE, down, msg_u))
end if
a =selectBestAction(actions)
executeAction(a)
```

to accept the downstream message. Otherwise, proposals from upstream are passed downstream, and from downstream to upstream. Note that messages will only be sent upstream if there is time for a reply to reach the client. The middleman may modify a proposal at this point (to make a profit) but this is omitted from the algorithm for clarity of presentation. Finally, the best action is selected and executed.

5.3 Experimental Evaluation

In the previous chapter, we presented an experimental evaluation of DiNE, our direct negotiation engine. This evaluation focused on core elements of the negotiation which would be present in any negotiations using DiNE. In this section, we evaluate the performance and behaviour of ChaNE, repeating the experiments in the previous chapter to examine any differences in behaviour, so that the implications of using chained negotiation can be determined.

In addition to the experiments performed previously, we examine behaviour traits specific to chained negotiation, such as the effect of middlemen taking a cut from negotiations. To evaluate the behaviour of ChaNE, we present four experiments: in the first experiment we vary the amount of time available in which to negotiate to observe the impact of chained negotiation; in the second experiment the number of middlemen is varied; in the third experiment we observe how chained negotiation behaves with multiple negotiation issues; and in the final experiment we examine the effect of the middleman making a profit by taking a cut of the proposals.

5.3.1 Experiment Setup

The experiments in this section all share the same basic structure as described in Section 4.3.1, where all variables controlling the outcome of a negotiation, such as the time available in which to negotiate, and the preferences of the parties involved, are grouped into an environment. Both the *end NCs* (the client and the supplier) use the same set of tactics described earlier (Boulware, linear and conceder polynomial time-dependent tactics). In each experiment, every tactic is run against every other tactic (including itself).

In a negotiation, clients and suppliers usually try to maximise their utility. Middlemen can exist to benefit the community they serve, or to make a profit by reselling items. We expect that when reusing existing commitments, clients receive a higher utility because of chained negotiation, as a favourable commitment may already be in place, instead of having to negotiate again with the supplier, whose preferences may have changed since the first agreement. However, it is harder to measure the benefit to the utility of the supplier, since it requires the assignment of utility to the case in which another client is satisfied without the supplier having to do anything. Instead, it is better to consider benefit in terms of being able to satisfy more clients and, consequently, we concentrate on client utility in these experiments. Comparisons are drawn again to the optimal utilities, calculated as described previously in Section 4.3.1.1.

5.3.2 Hypotheses and Results

5.3.2.1 Variable Negotiation Deadline

When an agreement must be in place by a certain time, deadlines are specified by which a negotiation must have completed. In direct negotiation, it was shown that a longer deadline leads to outcomes closer to the optimal utilities for each party, and shorter deadlines give a greater difference between the utility of the client and supplier. Shorter deadlines also increase the chance of a negotiation failing to make a deal.

In chained and forwarded negotiation, a single middleman is used for this experiment. Here, the amount of time taken for a message to be sent from the client to the publisher at the end of the chain is higher than in direct negotiation, which could lead to differing behaviour as deadlines are varied. In this experiment, we examine the behaviour of negotiation over a number of environments as the deadline is varied between 1 and 100 messages. We also record the amount of time taken in each negotiation.

Hypothesis: *With longer deadlines, chained negotiation produces utilities closer to the optimal set of values. As the deadline increases, chained negotiation completes in less time than forwarded and direct negotiation.*

Figure 5.7 shows the behaviour that results. First, direct negotiation increases above the optimal utility as soon as the deadline is long enough for a single negotiation. Then, it converges towards a level slightly lower than the optimal line in an oscillating manner, which is explained below. Forwarded negotiation follows a similar pattern, except that the period of the oscillation is greater, and convergence takes longer. Chained negotiation also oscillates, but this converges towards a significantly higher utility than direct and forwarded negotiations. This is because once a good commitment has been found, it can be reused for many subsequent negotiations. From Figure 5.8, it can be seen that the number of negotiations matched to existing commitments is very high, reusing commitments for most negotiations (over 95%).

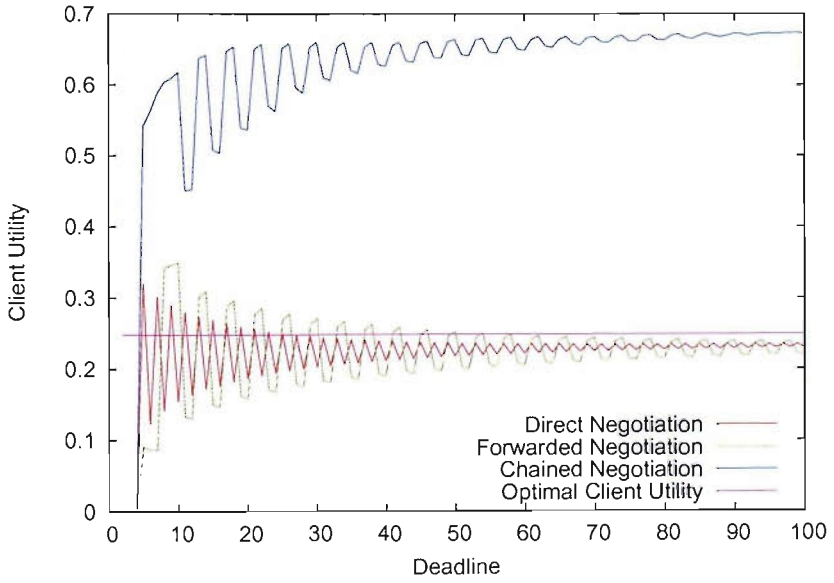

FIGURE 5.7: Client Utility of Direct, Forwarded and Chained negotiation as deadline is varied

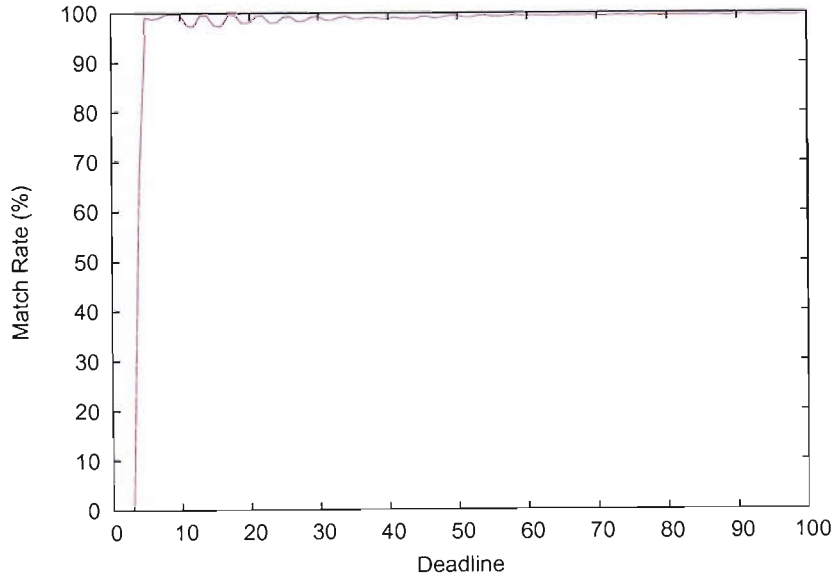

FIGURE 5.8: Match Rate for Chained Negotiation as deadline is varied

The oscillations in the utility of the consumer and supplier occur in each type of negotiation. To explain this effect, consider direct negotiation, where messages take a fixed amount of time to be transmitted (one period). Assuming that each party replies as soon as possible, it is possible to calculate which party is able to send the last message, as the deadline is measured in periods. This party will always be the one who offers their reservation value (the limit of the concessions they are prepared to make) if the negotiation has not completed by this point, indicating that they will receive the lowest possible utility for that deal. Thus, in direct negotiation, if the deadline is increased by 1, the two parties will swap over and the other will become the one to make the final concession, so that they receive the lower utility, causing the oscillations in the graph. Because in chained negotiation the time taken to send a message from client to supplier increases, the period of the oscillations is greater (twice the amount of time taken to send a message from client to supplier). The curves converge because where the negotiations complete before the deadline, more time in which to negotiate means that a mutually acceptable value closer to the ideal value can be found. For both direct and forwarded negotiations, if the supplier utility curve is plotted on the same graph, it oscillates at the same period, but out of phase with the client utility, as an increase in client utility comes at the expense of supplier utility. This behaviour was also seen in direct negotiation (Section 4.3.2.1).

Figure 5.9 shows the number of messages used in the negotiations in this experiment. On average, for any instance, direct negotiation and forwarded negotiation exchange the same number of proposals, linearly related to the deadline. Chained negotiation uses significantly fewer messages on average since, when an existing commitment is suitable, it is identified quickly. Thus we can conclude that once chained negotiation makes some initial commitments, further negotiations will take less time than direct and forwarded negotiations.

5.3.2.2 Number of Middlemen

The previous experiment examined the behaviour of the three types of negotiation as the deadline was increased, but using only a single middleman. Chained negotiation is likely to involve multiple middlemen, so we also need to examine behaviour as more middlemen are introduced.

Hypothesis: *As more middlemen are introduced, the utilities seen by the client and supplier will be further from the optimal. Additionally, the oscillations in utility will be larger due to the increased transmission times.*

Figure 5.10 shows that as the number of middlemen increases, the shape of the curve for the client utility with an increasing deadline does not change. However, the period of the oscillations of each curve increases as the number of middlemen is increased.

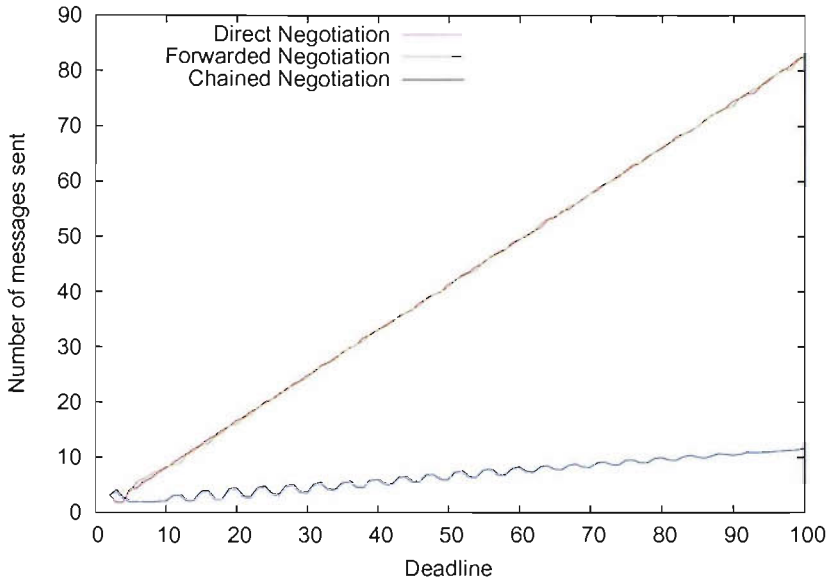

FIGURE 5.9: Number of messages sent in different negotiation types as the deadline is varied

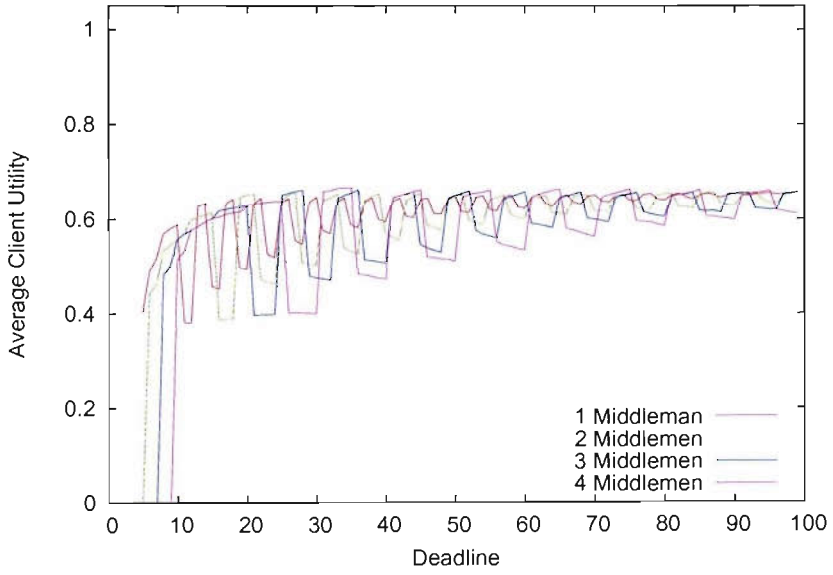

FIGURE 5.10: Utility of Chained negotiation as deadline is varied with multiple middlemen

This is due to the increased time it takes to send a proposal between the consumer and the supplier. However, with a larger number of middlemen, the client utility converges towards the same high value as with a single middleman, as in chained negotiation, existing commitments will be used where possible, and these are matched by the closest possible middleman.

5.3.2.3 Variable Number of Issues

When using a single issue, it is easy to match a new negotiation to an existing commitment. As more issues are introduced, the negotiation must find a proposal where all issues are satisfied by an existing commitment to avoid having to make a new one. In this experiment, we vary the number of issues in a proposal and determine how it affects the level of re-used commitments and utility received. A single middleman is used for forwarded and chained negotiation.

Hypothesis: *As the number of issues increases, chained negotiation satisfies fewer negotiations with existing commitments, but utility remains largely unaffected. Forwarded negotiation is not affected by the number of issues*

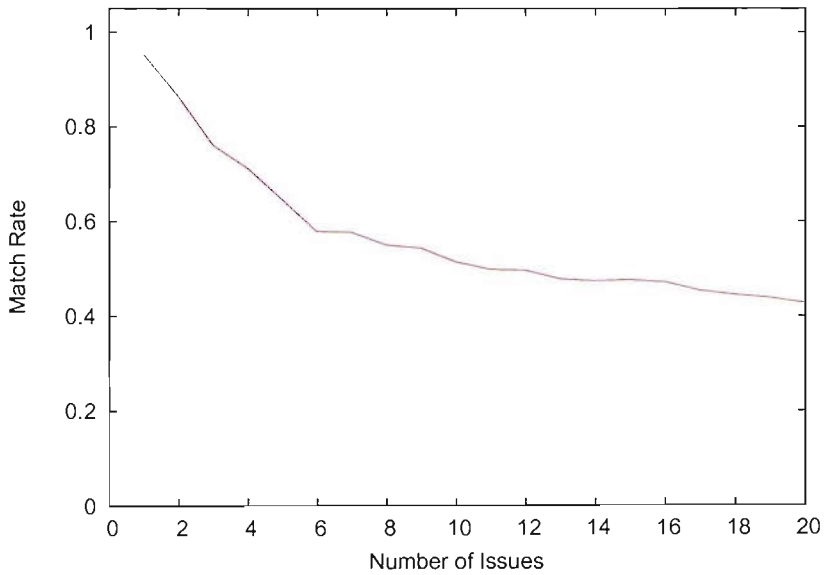

FIGURE 5.11: Amount of matched negotiations with variable number of issues

Figure 5.11 shows that in chained negotiation, as the number of issues is increased, the number of negotiations that can be satisfied using existing commitments decreases. Since matched negotiations generally have a higher client utility than unmatched ones, client utility decreases, but only converging towards the levels of forwarded and direct negotiation (as seen in Figure 5.12). If all issues are independent and concessions are made on each of them simultaneously, forwarded and direct negotiation are not affected by the number of issues. This is because the negotiation is constrained by the most restrictive issue rather than a combination of all issues.

Figure 5.12 also shows that as it gets harder for proposals to be matched by an existing commitment (i.e. as the number of issues increases), the performance of chained negotiation drops to the level of forwarded negotiation. From this, we can say that chained negotiation always performs at least as well as forwarded negotiation in this experiment. Chained negotiation will always perform at least as well as forwarded negotiation when

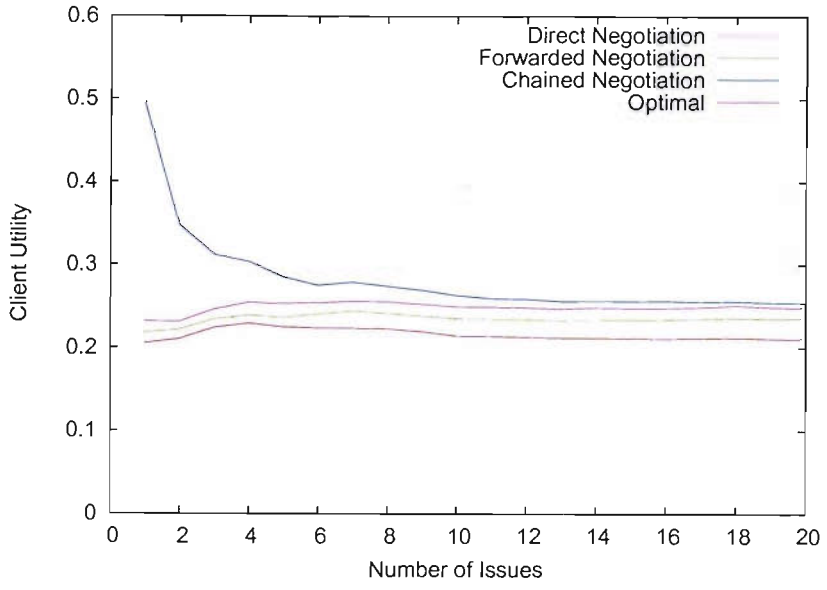

FIGURE 5.12: Client utilities in each type of negotiation with variable number of issues

attempting to match to existing commitments using the strategies we have described in this chapter.

5.3.2.4 Middleman profit rate

In both forwarded and chained negotiation, the middleman is able to modify a proposal before passing it onto the supplier, enabling the middleman to add a profit onto any cost issue in the proposal. However, if a middleman is making a profit, it becomes harder for the client and supplier to reach an agreement, and negotiations may fail. In this experiment, we examine the effect of a single middleman adding a profit to a proposal on forwarded negotiation. The proposals contain an issue representing the amount charged for the item.

Hypothesis: *As the middleman adds more profit, more negotiations will fail to reach an agreement. Additionally, the client utility and supplier utility will fall.*

Figure 5.13 shows that as the middleman adds an increasing amount onto a proposal as profit, the utility seen by both the client and the supplier decreases. As the profit rate reaches 60%, both utilities are very low. Not shown on the graph is the success rate — the percentage of negotiations that complete successfully, which decreases in the same way as utility, reaching 0 at 70% profit. This means that as the profit rate is increased, more negotiations will not reach an agreement, which is bad for all concerned. Also shown on the graph is the average profit made by the middleman per negotiation. Up to about 25%, this increases steadily. However, above 30%, too many negotiations fail, bringing the profit rate back down again.

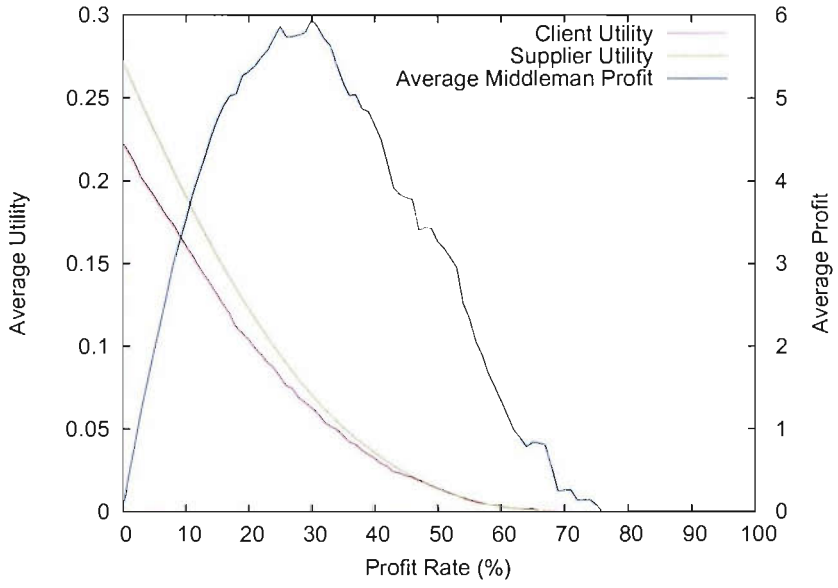

FIGURE 5.13: Utility and profit in forwarded negotiation with increasing profit rate

It is possible to combine these utility curves with the profit made by the middleman to calculate a utility to the system as a whole. To do this, we normalise the utilities of the client and supplier, so that their utility ($\bar{U_c}$ and $\bar{U_s}$) with no profit taken is 1, and create a utility function for the middleman

$$U_m = \frac{prof}{prof_{max}}$$

where $prof$ is the total amount of profit made by the middleman for that cut rate, and $prof_{max}$ is the maximum value of $prof$. We then combine these using a weighting ω ($0 <= \omega <= 0.5$) to form an additive utility function $\hat{U}$:

$$\hat{U} = (1 - 2\omega)U_m + 2\omega\bar{U_c}$$

Values of ω close to 0 favour the utility of the middleman, and values closer to 0.5 favour the utilities of the client and supplier. Figure 5.14 shows additive utility curves for different values of ω, and indicates that for all weightings, overall utility drops with a profit rate of above 30%. When favouring the utility of the middleman more, a profit rate of 20-30% gives a high overall utility. However, it can be seen than with any weighting, as the profit rate is set too high (above 40%), utility for the whole system begins to drop rapidly.

5.4 Sharing of Notifications

Chained negotiation in the context of a distributed notification service can reduce the degree of redundancy required in sending notifications, thereby increasing the number

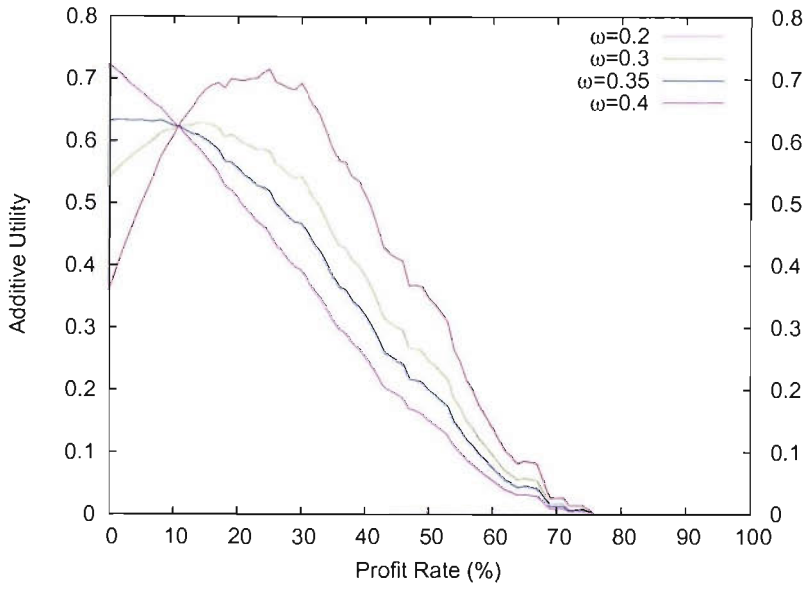

FIGURE 5.14: Additive Utilities with different weightings

of consumers a single publisher can serve. To show how this might happen, we set up a simulation shown in Figure 5.15, where 4 NSs are connected together in a chain, with the end of the chain being connected to a publisher. A large number of consumers are then spread evenly between the NSs and begin to request subscriptions with different parameters. Additionally, the publisher and each NS are assumed to be able to make a maximum of 500 commitments downstream, which leads to a theoretical optimum of 1997 consumers served (ns1,2,3 serving 499 consumers and 1 middleman each, and ns4 serving 500 consumers), with the publisher only having made a single commitment. This case is then contrasted with the case where all of the consumers communicate directly with the publisher. We then compare the number of consumers satisfied in each case.

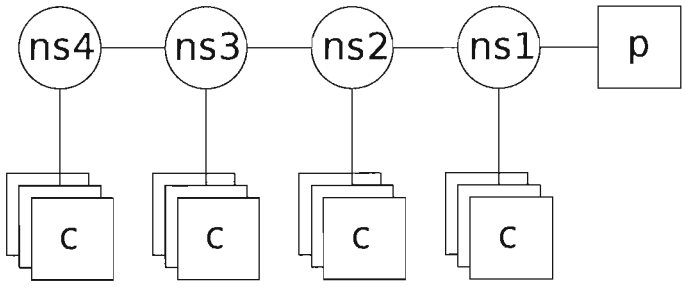

FIGURE 5.15: Consumers (c) connected via notification services (ns) to publishers (p)

As the publisher can only support 500 commitments, the case of direct negotiation showed that only 500 consumers could be supported. However, when the clients used their individual NSs using chained negotiations, many of the requests could be satisfied using existing subscriptions, *sharing* the notifications between consumers with similar requests. Here, 1911 consumers were satisfied before each NS reached their commitment limit, as shown in Figure 5.16. At this point, the publisher had only made 31

commitments, and was still able to satisfy many more consumers.

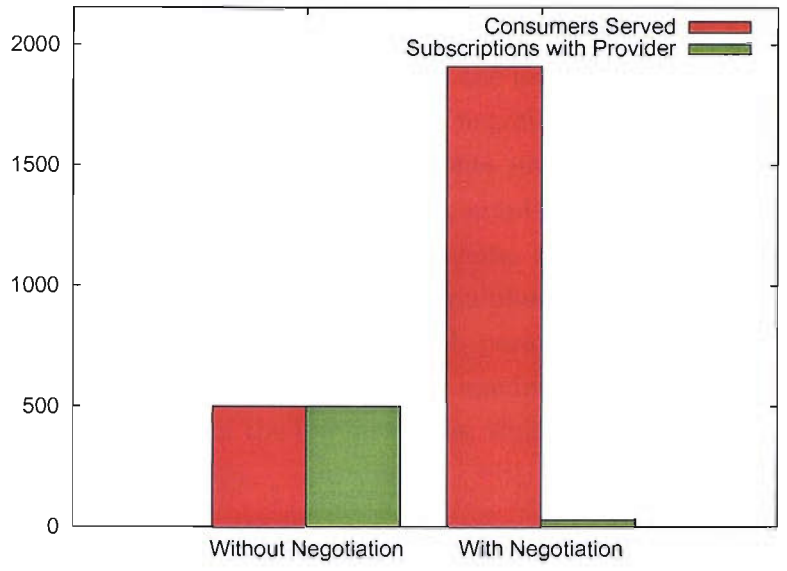

FIGURE 5.16: Consumers served and publisher's subscriptions with and without chained negotiation

The experiment was extended to allow middlemen to be arranged differently in a tree pattern, rather than the linear chain above. The arrangement was controlled by two parameters, d and w. These are illustrated in Figure 5.17. The depth of the tree is limited by the parameter d, the length of the chain between the provider and the furthest middlemen, and w limits the width of each node in the tree (the number of middlemen connected downstream of any given middleman). The provider and each of the middlemen were limited to making 100 commitments. The clients and providers used linear time-dependent tactics, and were run through 150,000 environments. The number of successful negotiations was counted and used as the metric for this experiment.

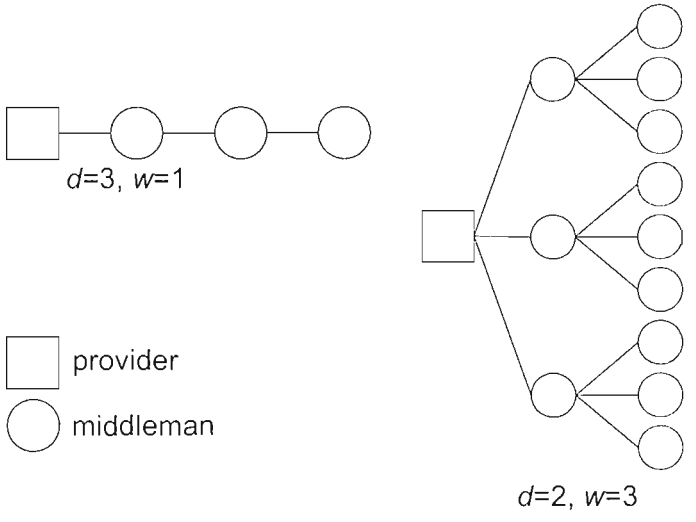

FIGURE 5.17: Examples of arrangement of middlemen

The comparison for this experiment is again the case where clients negotiate directly

with the provider, in this case only allowing 100 negotiations to complete successfully. With the middlemen arranged in a tree pattern, many more negotiations completed successfully. Figure 5.18 shows the number of consumers satisfied using different numbers of notification services acting as middlemen. As the number of middlemen at each node of the tree is increased, the number of successful negotiations increases. With $d = 1$, this increase is linear as each middleman only satisfies its own clients — each middleman makes the maximum amount of downstream commitments, while only requiring the publisher to make a small number of commitments, as shown in Figure 5.19. As the depth of the tree is increased, the increase in number of satisfied consumers becomes exponential as the number of child NSs at each parent NS increases. The number of commitments made by the publisher reaches its maximum with a lower number of child NSs per parent as the depth of the tree increases, while still allowing a large number of consumers to be satisfied.

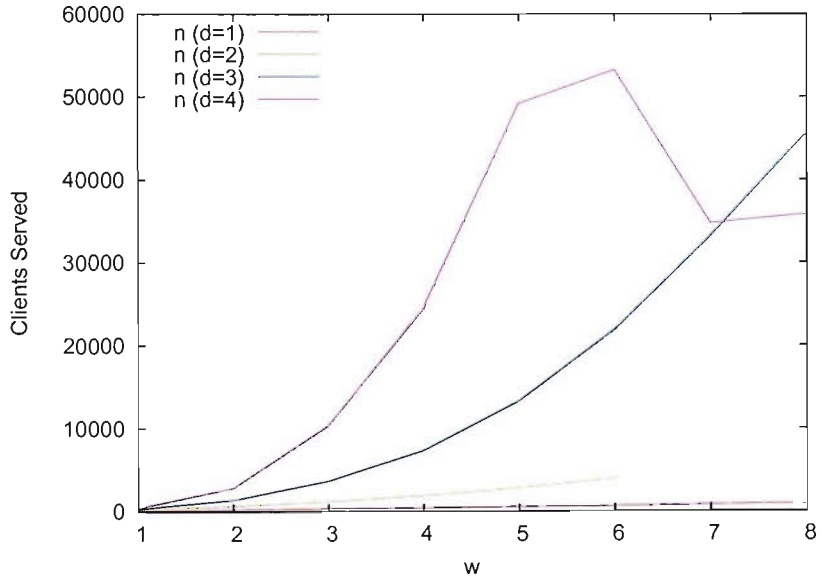

FIGURE 5.18: Consumers served with NS tree

However, as the number of overall middlemen in the system increases, a point is reached where the number of clients served starts to decrease, due to middlemen on the path to the supplier reaching their commitment limit. Chained negotiation works well when there are a large number of negotiations to form good commitments for other clients. When the number of middlemen is increased too far, the clients are spread around the tree too thinly to make and reuse good commitments efficiently. We checked this by repeating the experiment with a lower commitment level, restricting the middlemen and provider to 25 commitments. As shown in Figure 5.20, the decrease in number of clients served occurred with lower numbers of middlemen, proving this explanation.

Using negotiation in this scenario enables a single provider to serve significantly more consumers through middlemen than if they subscribing directly to the provider. The amount of increase in the number of consumers is controlled by the configuration of

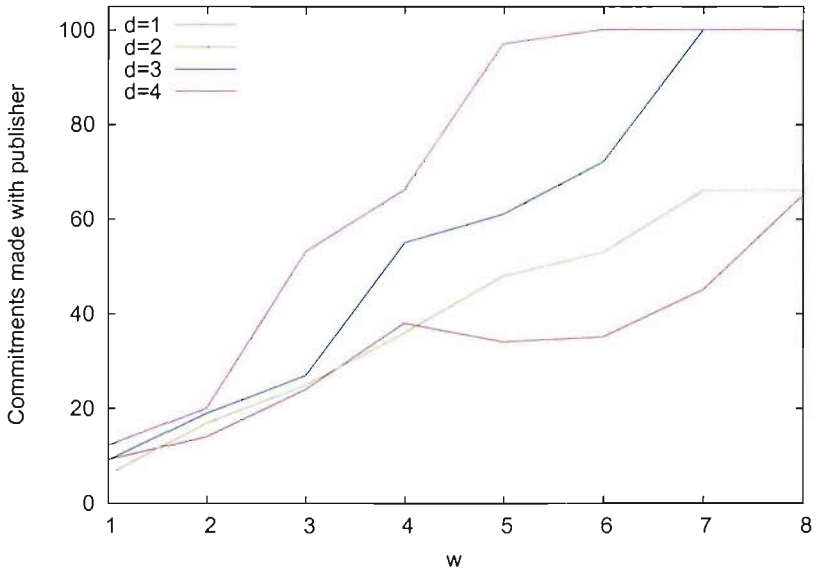

FIGURE 5.19: Commitments made with publisher in NS tree

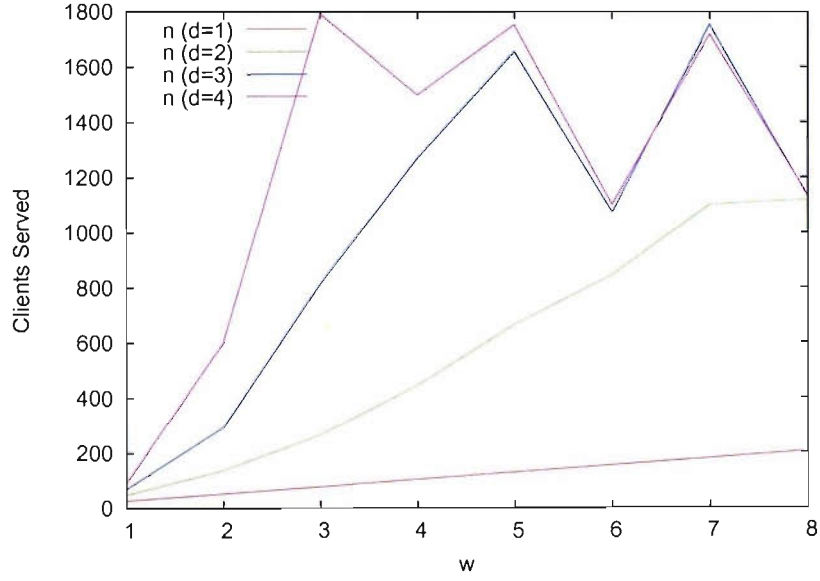

FIGURE 5.20: Consumers served with NS tree using lower capacity limit (25)

middlemen — a tree pattern gives the largest increase, but if the number of middlemen becomes significantly large, some clients may be precluded from making deals due to commitment limits being reached upstream of them.

5.5 Summary

In this chapter, we have presented the design of ChaNE, a Chained Negotiation Engine. This system represents an evolution of DiNE, the Direct Negotiation Engine presented in Chapter 4, and enables services to be made available with the possibility of being

redistributed or resold. The use of ChaNE in a notification service allows the services provided (i.e. notifications on a particular topic) to be shared amongst multiple consumers, *sharing subscriptions* between groups of consumers with similar interests, and enabling the NS to share notifications between then when a notification arrives.

We presented an evaluation of the behaviour of chained negotiation to determine its impact in comparison to direct negotiation. Chained negotiation introduces extra steps to the process of negotiation, so an evaluation demonstrating that the benefits of using chained negotiation outweigh the costs involved shows that it is a useful technique. These evaluations focused on three types of negotiation: direct negotiation (with no middlemen involved), forwarded negotiation (in which middlemen forward proposals without matching to existing commitments) and chained negotiation (in which middlemen forward proposals after attempting to match them to existing commitments). From the evaluation, we can make the following claims about chained negotiation:

- In all three types of negotiation, the outcomes are more predictable with a longer deadline than with shorter deadlines. When chained negotiation makes use of existing commitments, the results are significantly better for the client than direct and forwarded negotiation.
- Negotiations are always given deadlines by which to finish. With direct and forwarded negotiation, negotiations tend to use most of the time they have available. In chained negotiation, if an existing commitment can be reused, this is typically determined very quickly, so that chained negotiations take significantly less time to complete than direct or forwarded negotiations.
- Chained negotiation can in principle use an infinite number of middlemen. As the number of middlemen in a negotiation is increased, the costs of using chained negotiation also increase. With more middlemen, the minimum amount of time required for a forwarded negotiation is increased. Additionally, the utilities of the client and supplier differ more for a given deadline as more middlemen are used, and take more time to converge.
- Negotiations typically involve more than a single issue. Chained negotiation works best with a single issue. With few issues, chained negotiation still provides an increase in client utility. As the number of issues increases further, the amount of benefit decreases. However, even as the number of issues increases further, chained negotiation never gives a worse outcome than forwarded negotiation, which is not affected by the number of issues. Forwarded negotiation gives slightly poorer results than direct negotiation.
- If the middlemen in a negotiation make a profit by taking some of the cost issue in a proposal, this reduces the number of successful negotiations, and reduces the utility seen by both the client and supplier. If the profit margin is increased

too much, negotiations generally fail and nobody gains. Hence, a compromise is required between profit made by a middleman, and utility to the system as a whole.

ChaNE was designed for the context of a distributed notification service, where notification services attempt to share subscriptions to notifications on a particular topic between multiple consumers. To show the benefit of using chained negotiation in such a system, our simulation of the scenario showed that a significantly higher number of consumers could be served by the same provider using chained negotiation than with each of them making a subscription to the provider directly.

In summary, ChaNE enables services to be provided using chained negotiation, enabling the items of the service to be shared, redistributed or resold for the benefit of society as a whole, or for the benefit of the middlemen reselling the services.

The contributions in this chapter are the presentation of chained negotiation, and the evaluation of the chained negotiation engine. The chained negotiation model is novel as existing negotiation models do not incorporate input from intermediaries between a client and supplier. The evaluation of this model shows the behaviour of chained negotiation, and shows that by using such a model, a service provider can allow the redistribution or reselling of items enabling more clients to be served, and without imposing too much load on the service provider itself.

In the next chapter, we will describe the integration of ChaNE with a distributed notification service, creating an architecture in which a service provider can deliver its service through a notification service, negotiating with consumers over the QoS to provide the service, and having notification services share subscriptions to its content. We will show the benefit of the system by applying it to a practical application.

Chapter 6

QoS Negotiation in a Federated Notification Service

In Chapter 4, we showed that direct negotiation can enable a service provider to reduce the load placed on itself by finding a compromise between Quality of Service (QoS) requested by a client, and more manageable levels the service provider can maintain. Then in Chapter 5 we extended this negotiation model to develop *chained negotiation*, capable of negotiating between intermediaries who can also fulfil the negotiation by reselling or redistributing an item they have obtained from a previous commitment. In this chapter, we take ChaNE, the chained negotiation engine developed in Chapter 5 and integrate it with the myGrid notification service (MGNS), creating a new architecture where a service provider can support negotiation over QoS conditions for a service that is delivered using a distributed notification service. We then evaluate the benefits received by a real application adapted to this architecture.

6.1 Introduction

When a service provider allows a client to request a service with a particular set of QoS constraints, it is possible that a small number of clients requesting the service at a high QoS can place enough load on the service provider to prevent it from serving more clients. Different optimisation techniques can mitigate this effect: negotiation can be used to find a compromise between the levels of QoS a client requests and the levels a service provider considers manageable for a large number of clients, and sharing of the service output between clients asking for the same thing. A notification service achieves this by sharing subscriptions to a topic between all consumers connected to it that are interested in the same topic at the same levels of QoS. When a notification arrives on that particular topic, it is is shared between subscribed consumers.

QoS negotiation and sharing of subscriptions have both been examined separately in other work — QoS negotiation is commonplace in real-time and multimedia systems (Li and Ravindran, 2004; Rothermel et al., 1997), and existing distributed notifications reduce the number of redundant messages transmitted between NS instances, sharing subscriptions (Krishna et al., 2004; Pallickara and Fox, 2004a; Banavar et al., 1999a). In the previous chapter, we showed how *chained negotiation* could be used to combine the two optimisations, enabling consumers to negotiate over QoS conditions while still attempting to share subscriptions to notifications as much as possible, reducing the number of redundant messages sent in a distributed notification service.

Our contributions in this chapter are a presentation of a new architecture supporting quality of service negotiation with a service provider for services delivered over a distributed notification service, and the evaluation of the benefits provided a real application adapted to work in this architecture.

In this chapter, we present our approach to creating a negotiation-capable distributed notification service by integrating the chained negotiation engine into a notification service. In Section 2.6.1 we discuss our choice of notification services, then choose MGNS and explain that its current method for sharing notifications is not suitable for negotiation in Section 6.2.2. We then present our negotiation-capable version of MGNS in Section 6.2.3. In Section 6.3.1, we outline an application that can benefit from our NS, and adapt this to our architecture in Section 6.3.2. We then present experimental data in Section 6.3.3 to show the benefits this application received from the NS. Finally, we present a summary and conclusions in Section 6.4.

6.2 A Chained Negotiation-enabled Notification Service

6.2.1 Notification Services

In Section 2.6, we reviewed existing notification services, examining which of them employed subscription and notification sharing already, so that we may build on this to support QoS negotiation and sharing of notifications. Two of the NSs reviewed there, MGNS and NaradaBrokering, both seem to be suited to enabling negotiation over QoS while sharing subscriptions. In MGNS, when a consumer subscribes to a federated topic, the notification services set up subscriptions to each other to receive the notifications. Using negotiation, it would be possible for subscriptions to be set up only to those sources that are providing the relevant information at the desired QoS. In NaradaBrokering, the individual brokers in the network also set up subscriptions between each other to satisfy a consumer's subscription. However, due to the fact that NaradaBrokering allows a consumer to disconnect from one broker and reconnect to another broker at any point, setting up subscriptions based on a negotiated QoS would be harder to

efficiently maintain. For example, if a number of consumers all subscribe to the same topic at one location, so that only a single subscription is shared between them, they could then each move to different locations in the network. This means that each new broker must be subscribed to the topic, reducing the benefit from negotiating to get a similar level of QoS. Because of this additional complexity, MGNS has been chosen as a notification service to integrate with a negotiation mechanism.

In Elvin, the basic model is to flood the network with shared notifications and then send quench messages where these are not needed. This has been shown not to scale very well (Segall et al., 2000), so is not suitable for consideration here. Gryphon focuses on applying filters to notifications as far away from groups of consumers as possible, sending only the minimal amount of messages required. This technique provides an interesting opportunity to integrate with negotiation, but we choose to focus on resolving difference in QoS preferences instead of persuading clients to change their filtering preferences. The technique of multicasting notifications close to groups of consumers while filtering them as close to the source is also used in Siena.

6.2.2 Federated Topics in the myGrid notification service

The standard configuration of MGNS is a standalone notification service, with no facilities for interacting with other MGNS instances. However, MGNS can also be deployed as a distributed NS, in which it handles the distribution of messages between NS instances by using *federated topics* (Krishna et al., 2004). Normal topics in MGNS are unique within the scope of a single NS — publishers and consumers connect to the same NS in order to publish or consume messages on a specific topic. In a distributed deployment of MGNS, each NS instance operates independently of the others, continuing to use locally-scoped topics.

A federated topic is created using local topics at each NS. A local topic is created, and assigned some metadata giving the topic some semantic meaning. When semantic metadata is assigned to a local topic in a MGNS instance, the NS contacts a *topic registry* (Miles et al., 2003). A *virtual topic* record is then added to the registry, with a pointer to the local topic at a particular NS. When subsequent NSs have local topics created with the same semantic markup, they too register with the registry, adding their local topic as a *member topic* to the virtual topic record. Figure 6.1 shows a deployment of four MGNS instances, each having registered their local topics with the topic registry. In this example, NS-D is behind a firewall and cannot be contacted except through NS-C, so the topic registry also reflects this routing information.

In order for a consumer to subscribe to a virtual topic, the member topic at their local topic is looked up from the topic registry. The consumer then subscribes to this topic. The MGNS instance recognises that this is part of a virtual topic, and retrieves the

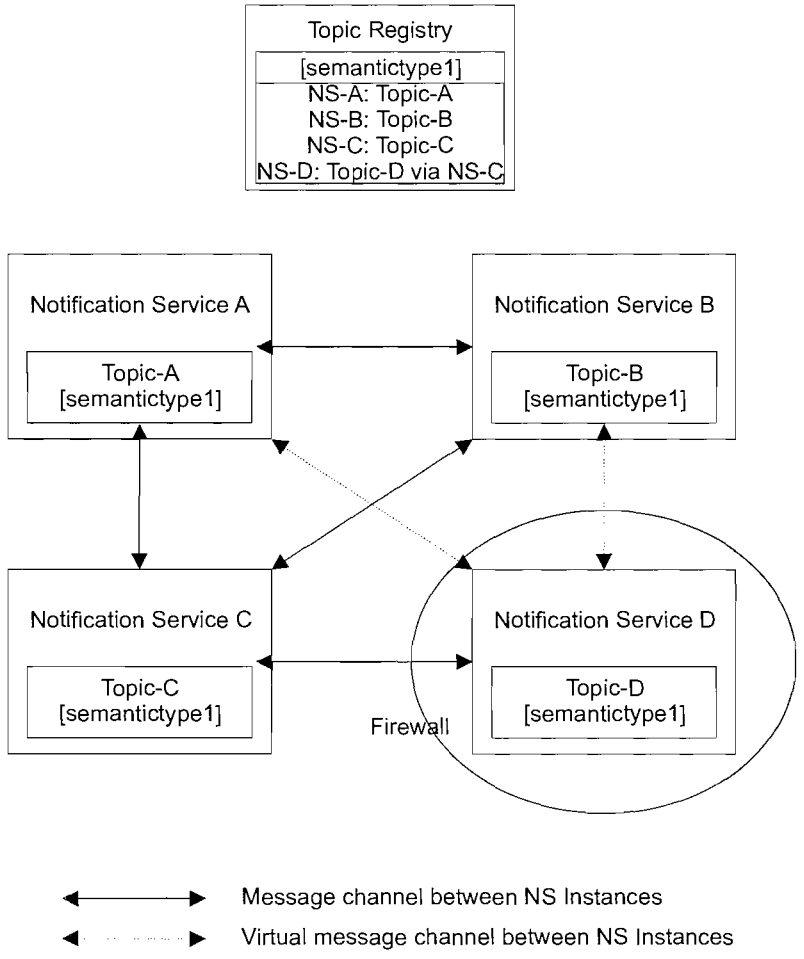

FIGURE 6.1: Federation of MGNS instances

virtual topic record from the topic registry. The MGNS instance then subscribes to the local member topics on each NS registered as part of the virtual topic, so that if a message is published on the virtual topic at a remote NS, every other NS will receive a copy of this notification and forward it to their consumers accordingly. In the example shown in Figure 6.1, the virtual topic record contains routing information to reach NS-D through NS-C. If a publisher publishes a message on this topic to NS-A, it knows that to forward this to NS-D it has to send the notification to NS-C, which will then forward on the notification. When messages are published on a virtual topic, the NS they are published to forwards the notification onto every other NS that has subscribed to the virtual topic. Message filters are used to prevent messages that have already been forwarded from being sent further (to avoid messages continuously being forwarded between the NSs).

This federated NS deployment enables consumers and publishers to be connected to different instances of MGNS in a network, and have notifications routed between them accordingly. A major disadvantage of this approach is that when there are more than two NSs participating in a virtual topic, the network is flooded by notifications whenever a message is published. Each NS pushes the NS received to every other NS, which

represents a potential scaling problem in a large-scale deployment. A solution where notifications are routed via intermediate notification services (similar to the sharing of subscriptions discussed previously in this thesis) could provide greater efficiency.

In its current form, the distributed version of MGNS is not suitable for integration with ChaNE for the following reasons:

- In MGNS federated topics, publishers can connect to any NS and publish messages. In chained negotiation, a consumer negotiates QoS conditions with the publisher. Chained negotiation is also used by intermediate NSs to set up shared subscriptions when multiple consumers subscribe to the same topic.
- MGNS only provides limited support for routing of notifications through intermediate NSs. Chained negotiation supports complex routes between publishers and consumers through a number of intermediate NSs, forming an efficient distribution network for publishing notifications.

For these reasons, we decided to base our integration of ChaNE into MGNS on a modified architecture for sharing of subscriptions between consumers rather than the existing federated topics. This architecture, which we refer to as ChaNNSe (**Cha**ined **N**egotiation-enabled **N**otification **Se**rvice), is described in the following section.

6.2.3 Integration of ChaNE and MGNS

To develop ChaNNSe, the method of sharing subscriptions between multiple consumers was replaced with a *subscription proxy*, which runs inside a MGNS instance. A subscription proxy subscribes to a topic on a remote NS, then republishes every notification received on that topic to the local NS, creating a *proxied topic*. Any number of consumers can be connected to the proxied topic, and they will all share a single subscription to the source of the notifications. The NS will share any notifications on that topic between them automatically. Figure 6.2a shows the notifications that are sent from NS-1 (where the message was published) to all other NSs participating in the virtual topic. In Figure 6.2b, the NSs use subscription proxies to limit the number of notifications sent between NSs. Consumers at NS-3 are subscribed via NS-2, so notifications are forwarded from NS-1 by NS-2, rather than an extra subscription being required. In this example, NS-4 has no interested consumers so is not receiving any notifications.

Figure 6.3 shows the components in a chained negotiation-enabled MGNS. Negotiation abilities are provided by a Negotiation Component (NC, described in Chapter 5), which has a communication module to implement communication between negotiation components via SOAP. When a consumer requests a service, it begins negotiating with the NC in the NS (1). The NC examines the *commitment store* to see if there are any existing

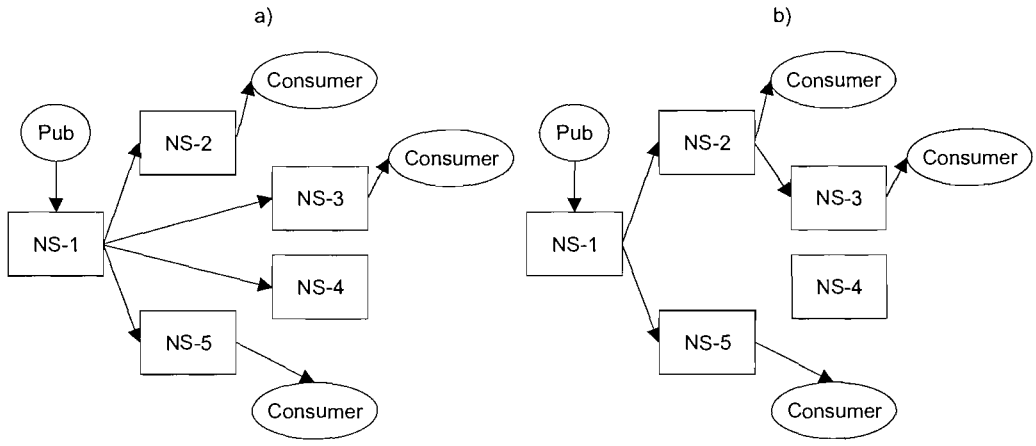

FIGURE 6.2: Sharing of subscriptions with a) Federated Topics, b) Subscription Proxy

subscriptions which will satisfy the request of the new consumer. If not, negotiation continues upstream (away from the consumer): either to another NS (2), or to a known service provider, which contains a NC, and a wrapper for the service it is providing. When a negotiation is successful with a service provider, the provider registers a topic with the nearest NS in the chain that it will use for delivering the results of the service. The NS then passes details of this topic back downstream. Any intermediate NSs add a proxied topic to their subscription proxy so that the results are fetched from the publisher's NS (4). The consumer then subscribes to the proxied topic at its local NS (5), and begins to receive results via the distributed NS (6). If any of the NSs in the negotiation chain are able to use an existing subscription, this is done instead of continuing to negotiate upstream.

Using this architecture, a service provider makes available a service with a variable level of QoS. A client interested in the service negotiates over the QoS via a number of intermediate NSs, which all attempt to provide the service the client is interested in without making additional subscriptions to the service provider. If the client and service provider reach an agreement on a set of conditions under which the service should be supplied, subscriptions are set up between NSs as needed to get the results of the service to the client. Notifications as a result of the service are shared between any clients with similar interests at the same NS if they can agree to use the same QoS levels already being provided by existing subscriptions. Otherwise, notifications cannot be shared.

In the next section, we describe an application suited to deployment in this architecture, and show how we adapt it to use ChaNNSe.

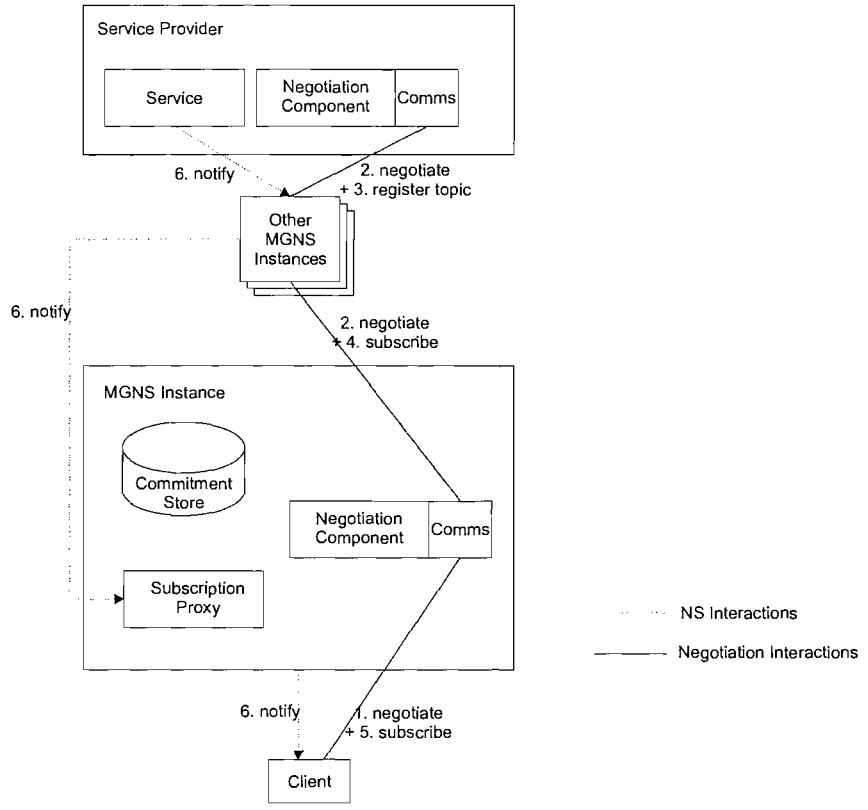

FIGURE 6.3: Components in ChaNNSe

6.3 Application and Evaluation of ChaNNSe

6.3.1 A Protein Compressibility Analysis application

Very large scale computations are now becoming routinely used as a methodology to undertake scientific research, examples of which can be found in many different fields. In Bioinformatics, a particular application of such an experiment is a *protein compressibility analysis* application, which uses a mixture of brute-force computation, statistical methods and guesswork in order to study the structure of protein sequences. The protein compressibility analysis application uses compression techniques to attempt to find patterns in protein sequences, and was designed by Zauner (Groth et al., 2005).

Proteins are the essential function components of all known forms of life; they are linear chains of typically a few hundred building blocks. Protein sequences are assembled following a code sequence represented by another polymer (mature mRNA), which is produced by splicing certain pieces of a molecular copy of the coding region of a gene on the DNA, while discarding other pieces of the copy. During the assembly, the protein curls up and forms a 3D shape, which determines its function.

The linear structure of a protein sequence is of interest for predicting which sections of DNA encode for proteins and for predicting and designing their 3D shape. For comparative studies of the structure present in a protein sequence, it is useful to determine

the textual compressibility of the sequence, as compression algorithms exploit context-dependent correlations within a sequence. The fraction of its original length to which a sequence can be compressed is an indication of the structure present in the sequence, but in general, no practical compression method can discover all of the structure in a sequence. Actual compression of a sequence can only yield a lower bound on its compressibility. For the same reason, compressibility values are also relative to the applied compression method. Methods that are good at discovering structure are computationally expensive; it is difficult to discover structure in protein sequences, although more progress has been made by grouping amino-acids: if the compression of the sequences is only for determining the structure, the sequences can be recoded with a reduced alphabet. For example, each amino acid symbol is replaced by a symbol representing a group of amino acids, and compression is then applied to the recoded sequence. The results of this analysis can then be used to determine the amino acid groupings that maximise compressibility.

This protein compressibility experiment is expressed as a workflow, as shown in Figure 6.4. A protein sequence sample is selected, potentially from several individual samples (Collate Sample). This is then recoded with a given group coding (Encode by Groups). The recoded sequence is then compressed with different compression algorithms (e.g. gzip, bzip2, ppmz) to obtain the length of the compressed sequence. Random permutations of the sequence (Shuffle) are also compressed to provide a standard for comparison, removing the influence of two factors from the calculation of compressibility: the data encoding used to represent the groups, and the non-uniform frequency of groups. From these results, a compressibility value is obtained for the sample sequence that is relative to both the compression method and group coding employed. The variability in the compressed length of the permuted sequences leads to a distribution of compressibility values (collate sizes). The workflow entails a sufficient number of compressions of permuted sequences to estimate the standard deviation for the compressibility (Average).

The measure step from Figure 6.4 is expanded and shown in Figure 6.5, where each sample is compressed using different compression algorithms, and the size of the compressed sample is measured. The size data from all of the measure steps is collated into a single table.

6.3.2 Adaptation of application for evaluation

To use the protein compressibility analysis application within ChaNNSe, the workflow shown in Figure 6.4 is run as the service in the service provider. In this scenario, scientists run the experiment repeatedly. The number of permutations used is considered a QoS condition, as a higher number of permutations gives a more accurate result. Hence the service is available with two QoS conditions:

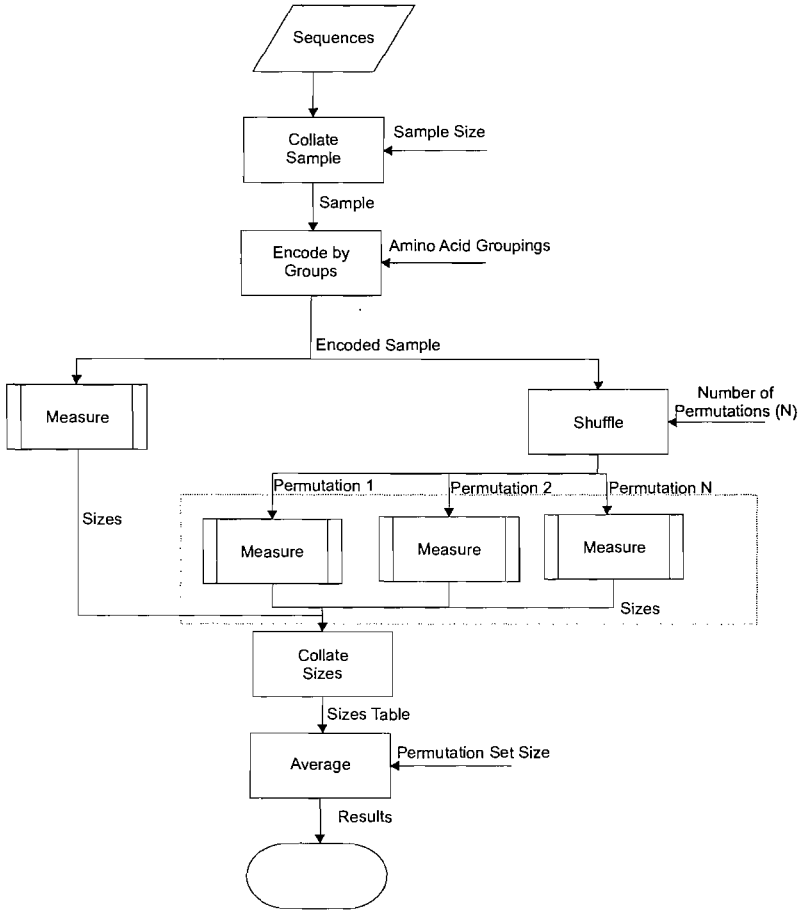

FIGURE 6.4: Protein Compressibility Workflow

- Frequency — The number of seconds between each consecutive run of the service for a particular client.
- Permutations — The number of permutations the input sequence is shuffled into during the experiment.

To be able to share notifications of the service running, consumers have to request the service with the same frequency and number of permutations. Every time the service is run, the output data is delivered to the consumer via the notification service. In this application, a commitment represents an accepted request to run the service with the specified intervals and the specified number of permutations. Each notification service involved in an accepted negotiation will subsequently hold one of these commitments which can be used for future requests.

6.3.3 Evaluation Process

ChaNNSe is intended to enable consumers and service providers to find a mutually acceptable QoS level under which a service can be provided, with the aim of reducing the

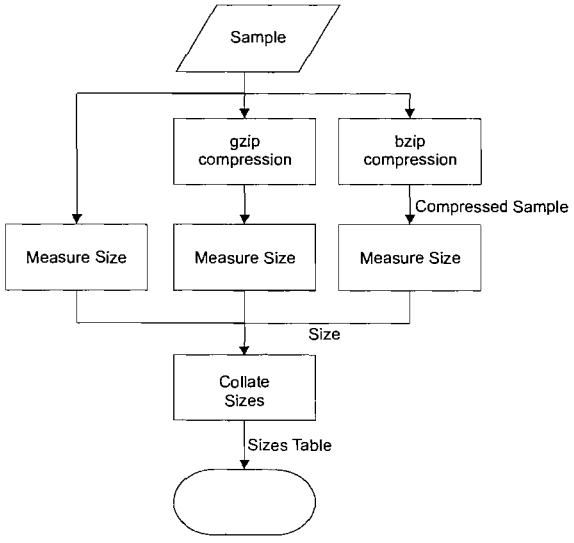

FIGURE 6.5: Protein Compressibility Measure Subworkflow

load on the service provider and enabling more consumers to be served by it. Through the sharing of notifications in the distributed notification service, more consumers can be served without even involving the service provider.

To determine how effective chained negotiation is in providing these benefits, we created a number of scenarios under which we could measure the load imposed by a number of consumers, and determine how many consumers can be served before the service provider is overwhelmed by the load. In each scenario, a single service provider is started with no existing commitments, and consumers request services from the provider until it is too busy handling existing commitments to accept new requests.

We examine the following scenarios, taking the same measurements for each:

1. *No negotiation, no shared subscriptions* — consumers request the experiment be run, supplying their ideal values for the frequency and number of permutations. The service provider accepts this request, without attempting to negotiate over the QoS levels.

2. *Negotiation, no shared subscriptions* — consumers request the experiment be run, and the service provider negotiates over the QoS. When a mutually acceptable set of conditions is found, the service is delivered to the consumers. Consumers cannot share subscriptions to the service.

3. *Negotiation, single NS sharing subscriptions* — the consumer and service provider negotiate over QoS as in the previous scenario, but the NS delivering the notifications allows consumers to share subscriptions, enabling multiple consumers with the same interest to only require a single subscription to the service provider.

4. *Negotiation, distributed NS sharing subscriptions* — as in the previous scenario,

negotiation and sharing of subscriptions are supported. However, multiple NSs are deployed in this scenario with consumers being spread between them.

In the experiments, we give the service provider a static set of preferences for both the number of permutations and the frequency with which to run the experiment. These preferences are shown in Table 6.1. The negotiation component for the service provider uses a single linear time-dependent tactic, as we would not expect such a detail to be varied on a service provider once running. The consumers will use either a Boulware, linear or conceder time-dependent tactic to represent different negotiation behaviour. Values for the deadline were chosen between 10 and 60 messages.

Issue	Ideal Value	Reservation Value
Frequency	43200	1800
Permutations	10	500

TABLE 6.1: Service Provider Preferences

For each scenario described above, the experiment comprises clients continually trying to subscribe to the service until the service provider, or a critical notification service on the path to the service provider, have reached their capacity. To determine the capacity of the service provider and notification services, we measure the CPU time taken to perform their relevant tasks and use this data in the experiment.

6.3.3.1 Service Provider Capacity

To determine the number of commitments a service provider can handle, the protein compressibility analysis application was run and timed. Due to the nature of the experiment, the number of permutations is the controlling factor in the amount of time the experiment takes to run. Hence, we measured the execution time of the application with values for the number of permutations between 50 and 1000. These measurements were run on an Intel Pentium 4 1.5GHz with 1Gb RAM running Debian Linux and Sun JDK 1.4.2.

Figure 6.6 shows that the time taken to run the protein compressibility analysis experiment increases approximately linearly. Hence, we model the relationship between permutations and execution time as:

$$CPUTime = Permutations * 0.174$$

As there are two variable factors for each request from a consumer, it is impossible to give an absolute limit on the number of jobs a service provider can handle. Instead, it is determined by the amount of CPU time that is used by the service provider executing the experiment the required amount in a day. As a simplification, we allow 80,000 seconds

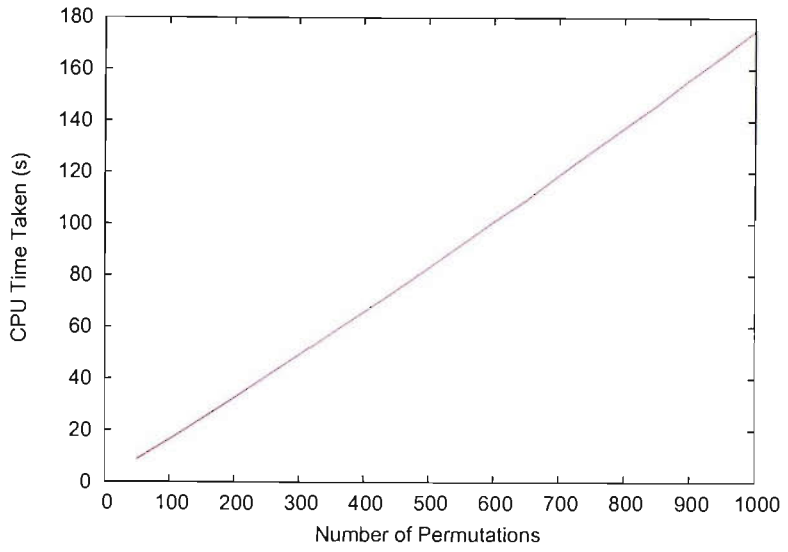

FIGURE 6.6: Execution time for Protein Compressibility Analysis application

of CPU time in one day, allowing the remainder for other tasks. The service provider can accept no more jobs when the following condition becomes true:

$$(MsgPerDay * AvPermutations * 0.174) > 80000$$

where $MsgPerDay$ is the number of messages that is required per day if a notification is sent out at the requested frequency for every consumer, and $AvPermutations$ is the average number of permutations for the jobs over the whole day.

6.3.3.2 Notification Service Capacity

To determine the limit a notification service can handle, we measured the time taken on MGNS for messages to be published and for them to be distributed to consumers. These measurements were taken using the same hardware as in the previous section. We found that publishing a message took an average of 60ms, and consumers pulling waiting messages[1] took 975ms. Both times were averaged over 100 iterations.

The limit that a notification service can handle is determined by the number of messages that it is sending and receiving. Hence, it is only affected by the number of consumers and the frequency with which they are receiving notifications. The notification service can no longer accept new requests when the following condition becomes true:

$$(UpstreamMsgPerDay * 0.06) + (DownstreamMsgPerDay * 0.975) > 80000$$

where $UpstreamMsgPerDay$ is the number of messages that are received from the NS

[1] Although we have used a pull model for consumers receiving notifications, a push model would be equally valid here.

or provider upstream, and $DownstreamMsgPerDay$ is the number of messages that are distributed to consumers or other notification services downstream.

It should be noted that the performance data for the NS may not be optimum, but this does not detract from the value of this experiment. Indeed, higher performance from the NS would only result in any benefit from using this architecture being magnified.

6.3.4 Results

For the consumers, their preferences were varied. These were calculated by randomly selecting a reservation value between the limits of the service provider's preferences, and choosing an ideal value in a range outside of service provider's preferences. This has the effect that for every negotiation it is possible to find a mutually acceptable set of QoS values, but every negotiation requires some negotiation in order to find this set.

For each scenario, we reset any commitments held by the service providers and NSs if applicable, then ran negotiations using a number of different consumers until either the service provider or a critical NS reached their capacity. In this context, a critical NS is one between a consumer and the provider in a chain — if a critical NS has reached capacity and a new subscription is required, the negotiation will fail. In each case, we measured the number of consumers served in total, the number of messages the service provider needs to send per day and the average value for the number of permutations that are used.

From the results in Table 6.2, we can see that in our reference case of no negotiation where a consumer is provided with the service at the QoS conditions they request, only 2 consumers could be served before the load they had placed on the service provider would prevent further commitments being made. When negotiation was used to determine more suitable QoS values, the number of consumers served increased significantly to 128. It can also be seen than in the case with negotiation, the service provider runs twice as many jobs as without negotiation, and that the number of permutations requested for each job is significantly lower, meaning that the amount of work required per job is lower.

Scenario/ Topology	Consumers Served	Provider Msg/Day	Average Permutations	Provider Load
1 /	2	659	610.2	87.5%
2 /	128	1296	347.3	97.9%
3 / -	2434	1143	399.7	99.4%
4 / Linear	1977	1044	437.0	99.2%
4 / Tree	1030	1076	427.0	99.9%

TABLE 6.2: Consumers served and service provider capacity

When shared subscriptions are introduced, the number of consumers satisfied by the

service provider increases dramatically. However, the number of messages sent per day by the service provider and the average number of permutations does not change significantly from the case where negotiation is used without sharing of subscriptions. This is because the additional consumers are served using a shared subscription at a NS, not requiring any additional intervention from the service provider. However, this scenario would place additional load on the NS. To check this, we measured the number of messages being received from upstream and being sent downstream by the NS. The results shown in Table 6.3 show that for the two scenarios without sharing of subscriptions, the NS handles a small number of messages, hence the load on it is very small. For the scenario using shared subscriptions, the NS sends nearly 10 times as many messages downstream as it receives from upstream. The load on the NS for this scenario is 63.3%, indicating that it is still possible for it to serve more consumers.

Scenario	Msgs Up	Msgs Down	Load
1	659	659	0.85%
2	1296	1296	1.68%
3	1155	50567	63.3%

TABLE 6.3: Messages sent by Notification Service for Scenarios 1-3

In the fourth scenario, chained negotiation using a distributed NS potentially enables a much larger number of consumers to be satisfied. Efficient distribution of notifications from the service provider could enable a very large number of consumers. In theory, if each NS runs at full load, the number of potential consumers that could be supported is determined only by the topology of the distributed NS. To show this, we compare the results from the topologies shown in Figure 6.7.

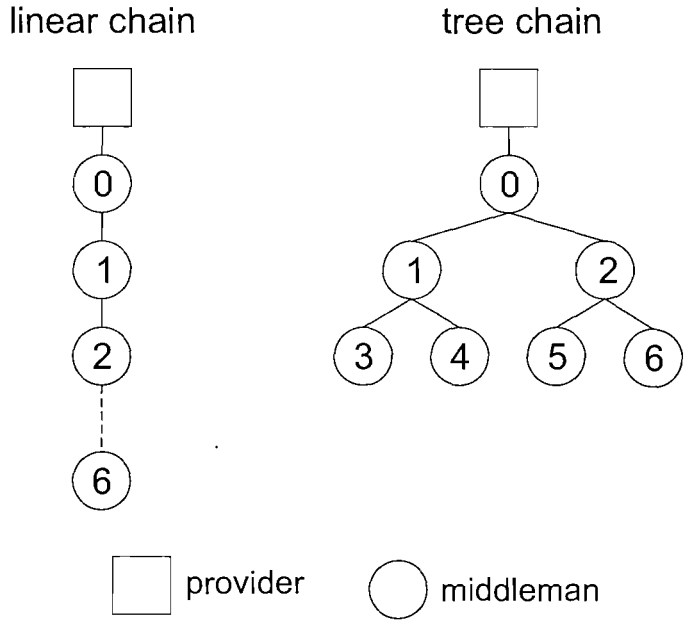

FIGURE 6.7: Topolgies in distributes NS for Scenario 4

The results from running the experiment for the different toplogies in Scenario 4 are

given in Table 6.4, along with a comparison result for Scenario 3. From this data, we can see that the distributed NS topologies serve fewer consumers in total than when using a single NS. In each case the service provider has reached full capacity.

Scenario/ Topology	Consumers Served	Provider Msg/Day	Average Permutations	Provider Load
3 / -	2434	1143	399.7	99.4%
4 / Linear	1977	1044	437.0	99.2%
4 / Tree	1030	1076	427.0	99.9%

TABLE 6.4: Consumers served and service provider capacity for Scenarios 3-4

Table 6.5 shows the number of messages sent by each NS in the toplogy and the load on each. Comparing this to the load on the single NS in Scenario 3, which received 1155 messages from upstream, sent 50567 downstream and ran at a capacity of 63.3%, we can see that the load has been distributed evenly around multiple NSs. In this example, there is no advantage to using a distributed topology in terms of numbers of consumers served. It does, however, distribute the load between NSs more evenly. From the results in Chapter 5, it might be expected that more consumers would be served using existing subscriptions. However, we showed in Section 5.3.2.3 that with more issues under negotiation, matching new requests to existing commitments becomes more difficult. This also explains the reason the tree topology serves fewer consumers than the linear one — as fewer consumers are going via each NS in the chain, the potential for matching to existing subscriptions is reduced. To show this more clearly, we ran the experiment again, using a fixed value for the number of permutations (we used 400, the average from Scenario 3), and only negotiating over the frequency. Table 6.6 shows that when only a single issue is used, the distributed NSs are able to serve many more consumers without having to place load on the service provider. In every case where subscriptions are shared, the service provider is not running at capacity and the experiment terminates because an NS has reached capacity. Table 6.7 shows the capacities of the NSs in Scenario 4, and shows that when deployed in a tree topology, the number of consumers that are satisfied is significantly higher than without a distributed NS, or with a linear topology of NSs.

Topology	Linear			Tree		
Middleman	US	DS	Load	US	DS	Load
0	1204	7302	8.99%	1078	4254	5.27%
1	1287	7073	8.72%	711	3633	4.48%
2	1288	8429	10.4%	757	3883	4.79%
3	1153	7943	9.77%	364	3024	3.71%
4	932	7986	9.80%	475	3222	3.96%
5	673	7188	8.81%	336	3006	3.68%
6	494	6789	8.31%	410	2947	3.62%

TABLE 6.5: Messages upstream (US), downstream (DS) and NS Capacities (Load) for Scenario 4

Scenario/ Topology	Consumers Served	Provider Msg/Day	Average Permutations	Provider Load
1 / -	6	1124	400	97.8%
2 / -	125	1146	400	99.7%
3 / -	2956	137	400	11.9%
4 / Linear	2749	179	400	15.5%
4 / Tree	20488	519	400	45.2%

TABLE 6.6: Consumers served and service provider capacity (Single Issue only)

Topology	Linear			Tree		
Middleman	US	DS	Load	US	DS	Load
0	184	12099	14.8%	519	82019	100%
1	176	12177	14.9%	338	82029	100%
2	223	10895	13.3%	395	80760	98.5%
3	20	1152	1.41%	241	82036	100%
4	22	3392	4.14%	233	82036	100%
5	77293	77293	100%	315	78674	95.9%
6	58758	58758	76.0%	173	82040	100%

TABLE 6.7: Messages upstream (US), downstream (DS) and NS Capacities (Load) for Scenario 4 (Single Issue only)

6.4 Summary

In this chapter, we have described ChaNNSe, an architecture for QoS-aware service providers to deliver their services through a distributed notification service. We implemented this by integrating ChaNE, the chained negotiation engine described in Chapter 5 with the myGrid notification service. We adapted an application using a protein compressibility analysis to be used in this architecture, and presented an evaluation of the performance benefits received from using it.

From the evaluation of ChaNNSe presented in this chapter, we can make the following statements about the use of chained negotiation in a distributed notification service:

- A service provider allowing QoS to be selected by its clients risks running itself to full capacity for just a few clients if it does not attempt to negotiate the QoS level with the client. However, by supporting negotiation, the number of clients it can support increases significantly.
- If a service provider is likely to be offering the same service to multiple clients, it is a waste of time to do the same job twice. By using a notification service to deliver the service to consumers requesting the same service, the service provider only need carry out the job once. This significantly increases the number of consumers that can be served.
- A notification service itself can reach capacity by having many consumers request-

ing a service through it. To combat this, distributed notification services are used, with consumers able to connect to different instances of a NS. Using chained negotiation and the distributed NS, they can share subscriptions to a service with other consumers at the same NS without imposing additional load on the service provider. Notifications are then shared between the consumers whenever one is received on the shared subscription. Different choices of distributed NS topology allow the load to be spread between multiple NSs, removing potential bottlenecks.

In summary, using chained negotiation in a distributed notification service enables a service provider to serve more clients than by dealing with them directly. This is due to two reasons — chained negotiation is used to determine QoS conditions acceptable to both the service provider and the client, and the distributed notification service shares subscriptions to the service provider between as many consumers with similar interests as possible. This results in an efficient distribution system for the service provided by the service provider.

The contributions in this chapter are the presentation of a new architecture supporting quality of service negotiation with a service provider for services delivered over a distributed notification service, and the evaluation of the benefits provided a real application adapted to work in this architecture.

Chapter 7

Conclusions & Future Work

7.1 Summary

In this thesis, we have presented an architecture enabling service providers to make their service available through a distributed notification service (NS), efficiently delivering notifications to consumers by sharing subscriptions to notifications between consumers with similar interests. Consumers and service providers negotiate to find mutually acceptable quality of service (QoS) levels under which the service can be provided — high enough to satisfy the consumer, without placing unnecessary load on the service provider.

In more detail, we have reviewed existing work in the field of notification services, paying attention to how distributed notification services share subscriptions and notifications amongst multiple consumers with similar interests. A NS shared subscriptions to a topic when multiple consumers request the same topic with the same QoS conditions. Notifications are shared between them when one is received on that topic. We then reviewed work in the field of automated negotiation, examining different negotiation mechanisms used both in automated and non-computational negotiation. After examining different automated negotiation techniques, we selected a heuristic-based approach upon which to base our negotiation engines.

We presented DiNE, the first evolution of our negotiation engine, supporting direct negotiation (between two parties only). We evaluated the behaviour of DiNE in performing direct negotiations to determine the effect of different deadlines and different numbers of issues under negotiation.

We then discussed the need for a new type of negotiation involving intermediaries, and introduced chained negotiation for this task. ChaNE, the next evolution of our negotiation engine which supported chained negotiation was presented along with an evaluation of its behaviour, showing the benefits that chained negotiation can offer.

To validate the application of chained negotiation to a distributed NS, we have developed ChaNNSe, a chained negotiation-enabled NS based on ChaNE and MGNS. This enables a service provider to allow consumers to request QoS conditions for a service, without placing unnecessary burden on the service provider. It also enables the efficient delivery of notifications for the service by sharing subscriptions to notifications between consumers at a NS with similar interests, further helping to reduce the load on a service provider.

7.2 Research Contributions and Publications

The contributions of this thesis have been:

1. We have shown that by using negotiation, a service provider can enable a client to request a service with a specific Quality of Service that is high enough to satisfy the client, and manageable enough so that it will not place an unnecessary load on the service provider, enabling it to support a large number of clients. We created a direct negotiation engine and showed through simulations that it would allow QoS levels to be requested along with a subscription in a notification service. Existing notification services allow QoS conditions to be specified, but make no attempt to resolve differences between the preferences of the client and supplier. Our empirical evaluation of our negotiation engine has shown the behaviour of the system independent of external influences.

2. As direct negotiation is unsuitable for negotiation through intermediaries, we developed *chained negotiation*, a new form of negotiation that involves the client and supplier as well as any number of intermediaries between them. Intermediaries can fulfil a client's request using an existing commitment they have made on behalf of an existing client, redistributing or potentially reselling the item. We have used this as the basis for ChaNE, a chained negotiation engine. The evaluation of ChaNE shows the benefits that can be obtained by using chained negotiation in a distributed notification service. Chained negotiation is a novel form of negotiation — existing forms of negotiation only involve clients and suppliers.

3. We have taken our chained negotiation engine and integrated it with an existing notification service, creating ChaNNSe, a novel architecture for supplying services over a distributed notification service while enabling negotiation over QoS to resolve differences between the consumer and service provider. We have used a real application from the field of Bioinformatics to show the benefits offered by this architecture. As stated earlier, existing notification services do not support negotiation over QoS conditions. Hence, this represents a novel contribution.

The work on direct negotiation was presented at the Ninth International EUROPAR conference (EURO-PAR '03) in Klagenfurt, Austria (Lawley et al., 2003a) and published in a special issue of Parallel Processing Letters (Lawley et al., 2003b).

Chained negotiation was presented at two different conferences: the second UK e-Science All Hands Meeting (AHM '04) in Nottingham (Lawley et al., 2004); and the IEEE/WIC/ACM International conference on Web Intelligence (WI '05) in Compiègne, France (Lawley et al., 2005).

7.3 Conclusions

When a service provider enables clients to request certain QoS conditions under which a service should be provided, there may be differences between levels of QoS the client desires, and levels a service provider can maintain for a large number of clients. Negotiation can be used to resolve these differences, finding a compromise between the preferences of both parties that is mutually acceptable. DiNE, our direct negotiation engine, enables automatically finding this compromise. Through our evaluations, we have identified the following behavioural characteristics of DiNE:

- Negotiations are often given a deadline by which to complete, in order to ensure a service is delivered when it is needed. When short deadlines are used, the outcome of the negotiation is harder to predict than with a longer deadline. Concessions must be made rapidly in order to reach an agreement before the deadline expires, hence the outcome tends to be further from an optimal solution for both parties.
- A negotiation usually involves multiple issues, potentially trading one issue off against another to find a mutually acceptable solution. A large number of issues makes no significant difference to the outcome of negotiations, assuming each issue is independent from the others.
- If it is possible to predict the message transmission times, it may be possible for a client to choose a deadline such that it can force the supplier to offer the final proposal in the negotiation, typically causing it to offer its reservation value. The supplier's reservation value represents the lowest possible utility it can receive, and the highest possible utility the client can receive. However, it would be difficult in a real implementation to accurately predict this message transmission time.

Direct negotiation, however, does not enable intermediaries between a client and a supplier to provide input to the negotiation process. In a distributed NS, intermediate NSs may be able to fulfil a consumer's request by offering to share an existing subscription created for another consumer. Chained negotiation enables intermediaries to participate

in the negotiation, allowing the reselling or redistribution of items to clients. We developed ChaNE to support chained negotiation, and through its evaluation we can make the following statements about chained negotiation:

- As with direct negotiation, longer negotiation deadlines give results closer to the optimal solutions when no existing commitments are available. However, once favourable commitments have been made by a middleman, clients that can make use of them complete negotiations very quickly, getting a higher utility than continuing to negotiate with the supplier and not requiring the supplier to do any additional work.
- As chained negotiation increases message transmission times, it has the effect of reducing the amount of time available in a negotiation. As more intermediaries are used, this effect is magnified, increasing the minimum amount of time required for a negotiation and making negotiations take longer to converge.
- As the number of issues in a negotiation is increased, it gets harder for a middleman to match a new request to existing commitments, as it must match on all issues. Hence as the number of issues is increased, the benefits of chained negotiation are decreased, eventually falling to the same level as if existing commitments were not available.
- Intermediaries in a chained negotiation can make a profit by adjusting a cost element within proposals in order to take a portion of the reward for supplying a service. However, this process makes it more likely that the negotiation will fail to find a mutually acceptable set of conditions under which the service can be supplied. If an intermediary tries to make a large enough profit, very few negotiations will succeed and all participants lose out.

The objective of this thesis has been to enable a service provider to make its service available to consumers who can request levels of QoS, while still enabling the efficient delivery of the service to consumers and allowing the service provider to support as many clients as possible. Our solution to this problem was to integrate chained negotiation into a distributed NS, allowing consumer and service provider to negotiate over QoS levels for a service, and allowing the distributed NS to share subscriptions to notifications from the service to be shared amongst consumers with similar interests. From evaluating this system, we can make the following statements:

- Chained negotiation enables a consumer and publisher to find a mutually acceptable set of conditions under which a service can be supplied, while still enabling a NS to share subscriptions to the service.

- By supporting negotiation between the consumer and service provider, the consumers do not always get the high QoS they would ideally like, but do get a level they are prepared to accept that will not place unnecessary burden on the service provider, which could prevent the service provider from being available to subsequent consumers.

- By sharing subscriptions to notifications from the service, the distributed NS further reduces the load placed on a service provider by supporting additional consumers without any extra burden being placed on the supplier. This also reduces the load placed on the distributed NS itself, as fewer notifications have to be sent between the individual NSs.

- As the number of instances in a distributed NS is increased, the number of potential consumers that can be satisfied by a single service provider can increase dramatically. Different topologies of a distributed NS can offer even better results — for example a tree-style topology enables a higher potential number of consumers than a linear topology. However, as the distance between consumer and service provider increases, the performance of chained negotiation deteriorates. As more issues are negotiated over, it becomes harder to match a new request to an existing commitment (thereby reusing a shared subscription). Hence, a balance must be found between the topology of a distributed NS and the different benefits it offers.

Overall, we can say that by using chained negotiation in a distributed notification service, a service provider can enable clients to request services with a specified QoS and have differences between the preferences of the client and provider resolved, while still enabling the distributed notification service to share subscriptions to the service, sharing notifications between groups of consumers with similar interests.

7.4 Limitations

Although we have demonstrated in this thesis that chained negotiation can provide a benefit to a service provider by enabling it to negotiate over QoS with a client, and share subscriptions to its service through a distributed notification service, we recognise that there are a number of limitations:

- In both direct and chained negotiation, the negotiation issues must be independent. In a real life implementation, it is likely that at least some issues will be dependent on others, for example if a service is going to be run with a higher priority, the service provider would expect to be able to charge more for it. Dependent issues are more complex to model, and lead to the notion of trading one issue off against

another. Due to the complexity of dependent issues, adding support for them would have required a significant effort, and was hence considered beyond the scope of this work.

- The evaluations of both direct and chained negotiation have used a single tactic for proposal generation, and single linear utility functions for proposal evaluation. In a real implementation, it is highly likely that multiple tactics would be used, and potentially more complex utility functions. However, it was important to evaluate these parts of the system independently of external influences to determine their behaviour. As resource-dependent tactics are domain-specific in nature, it would have been impossible to evaluate them independently of a specific domain. The evaluations also used an interval-based model of time where exactly one message is sent in each interval, rather than a real-time based model. Using real time would have meant that additional variables, such as the amount of time required to compose and send a message (which will most likely not be constant) must be introduced.

- The integration of chained negotiation into a distributed notification service has focussed on the context of a service provider delivering its service through a notification service, rather than a general notification service environment where there can be multiple publishers and multiple consumers all attached to the same topic.

- In some distributed notification service (e.g. NaradaBrokering), consumers may disconnect from a broker and reconnect to another point in the network, and still have notifications routed to them. ChaNNSe does not support this relocation as chained negotiation sets up subscriptions for notifications to be delivered along a fixed route, but it may be possible to dynamically change the subscriptions to follow the consumer.

7.5 Future Work

While we have demonstrated that chained negotiation can provide significant benefits to the provision of services in a distributed notification service, we have also identified a number of areas for potential improvement and for further future work.

7.5.1 Real-time Deadlines

Both DiNE and ChaNE used deadlines based on the number of messages that can be transmitted during the negotiation. This approach enables chained negotiation components to accurately determine whether they have enough time to negotiate upstream, a crucial part of chained negotiation. In a real implementation, it may sometimes be

desirable to specify a deadline in terms of an absolute time, rather than how many messages can be sent in the negotiation. This would be needed in a real-time application where a client is negotiating for provision of a time-critical service.

To convert the negotiation engines to use real-time, some issues must be overcome. The most important issue is that a mechanism must be created to enable each negotiation component to determine whether it has enough remaining time for a negotiation to proceed upstream. If it cannot accurately determine this, a message may be sent upstream without enough time for the reply to reach the client before the negotiation deadline expires, causing the negotiation to fail outright. One possible method for doing this would be to measure the average time taken for a message to be sent over the course of each negotiation, and use this average to determine the number of messages that can be sent in the remaining time. Problems such as network latency and irregular traffic might cause these numbers to become distorted.

When using real time, the problem of potentially predicting which party will make the final proposal in a negotiation (discussed in Section 4.3.2.1 on Page 61) will be reduced. As message transmission time is likely to be variable, it may be impossible to accurately predict which party will make the final concession.

7.5.2 Parallel Negotiations

Chained negotiation can be seen as a chain of individual negotiations. A middleman is a participant in two negotiations at once, upstream (towards the supplier) and downstream (towards the client). In our implementation of ChaNE, negotiation proceeds in one direction at a time. It may be possible to conclude negotiations quicker if a middleman is able to negotiate in both directions simultaneously, making concessions with both sides of the negotiation.

Additionally, a middleman may be able to recognise that two clients are requesting the same item at the same time, and can attempt to combine the two before they complete, potentially getting a better deal from the supplier than if it were negotiating for one client only.

7.5.3 Renegotiation of existing commitments and combining of commitments

With the current implementation of ChaNNSe, a NS will attempt to share a subscription to a consumer that requests a similar set of QoS conditions. However, if the consumer requests a higher or incompatible set of conditions, it will be impossible to share the subscription and a new one may be made. In this situation, it would be possible to examine the current set of subscriptions and determine if any of them are unnecessary.

For example, in a service providing stock quote updates with a specified frequency, consumer A may request updates for a particular stock every 2 hours. If consumer B then requests updates for the same stock, but every hour, the NS could cancel the initial subscription and use the 1 hour subscription to satisfy both clients — it could forward the notifications immediately to consumer B, and store alternate notifications to deliver a combined notification to consumer A.

Similarly, a NS could learn the common interests of its consumers, and factor this information in when making subscriptions. For example, if one service proves very popular with many consumers, the NS could negotiate with a service provider for a higher QoS, justifying it by saying it will support multiple consumers using the provided subscription. The NS can also do extra work to translate or filter the data provided by the publisher in order to satisfy multiple consumers — taking a single subscription that has enough content to satisfy all of its consumers and doing additional translations or filtering as appropriate to each consumer. Mühl (2002) describes a similar idea applicable to content-based publish/subscribe systems where the content of notifications is determined by filters specified by consumers — NSs can share a subscription by using algorithms to combine filters, requesting a single subscription from the publisher containing enough information to satisfy multiple consumers.

A third related possibility is to enable middlemen to use multiple existing commitments to satisfy a new request. For example, if consumer A has requested notifications every 2 hours (odd numbered hours), consumer B has requested notifications every 2 hours (even numbered hours), when consumer C requests hourly notifications, a NS could use the commitments held for both consumer A and B in order to satisfy C.

7.5.4 Protocol compliance and agreement monitoring

A negotiation protocol specifies the rules which a participant in a negotiation should adhere to. However, it is possible for a malicious agent to break some of the rules in chained negotiation to get a better outcome at the expense of its opponent. For example, messages in chained negotiation contain a field indicating the number of hops from each end of the negotiation. By falsely setting one of these values, it would be possible for an agent to convince its opponent that it should prematurely offer its reservation value. This would cause the agent offering the reservation value to get the minimum possible utility for that negotiation, while the malicious agent gets the maximum utility. It is also possible for middlemen to accept proposals from clients without actually having made arrangements to supply the item. A malicious middleman would take payments for a service and then not supply it. It would be desirable for some checking or enforcement of the rules in the protocol so that it becomes harder or impossible to break them without being detected.

Approaches using cryptographic techniques could be used to ensure the number of hops in the message is accurate. Alternatively, random checks could be carried out where the client makes direct contact with the supplier to determine if protocol rules have been broken by an intermediary. If an agent is found to have broken the protocol rules, a negative reputation could be left. We believe that a trust model or reputation mechanism, such as those presented in (Patel et al., 2005) and (Zacharia et al., 1999) would be of use in this situation.

Related to the area of protocol compliance monitoring is agreement monitoring. In ChaNNSe, consumers can request a particular quality of service when they request a service. Especially in cases where the consumer has paid for the service, it is desirable to have a mechanism for monitoring the quality of service actually received by the consumer. The WS-Agreement specification provides an interface for monitoring of agreements, and (Ludwig et al., 2004) details an implementation of this work. Monitoring of agreements using such techniques would be of benefit to the ChaNNSe architecture.

7.5.5 Further negotiation techniques

ChaNE uses a heuristic-based approach to automated negotiation. However, we believe that some additional negotiation techniques may be of use in improving the performance or providing additional functionality. For example, the use of argumentation-based negotiation may be helpful to the extra work on renegotiation suggested earlier — a middleman can negotiate for higher QoS than would normally be received from a supplier by justifying the need for the higher quality of service as being required to support multiple clients. It may also help improve the performance of chained negotiation — if a supplier can make no further commitments, a middleman may suggest an existing commitment and inform the client that this is being suggested because the supplier cannot satisfy the request directly.

7.5.6 Compliance with Industry Standards

ChaNNSe is based upon an automated negotiation engine and a notification service. Both of these fields have active development of a web standard — WS-Agreement for negotiation and WS-Notification for notification services. At the time of writing, neither of these standards had made it to a final version, so building in support for them was considered unnecessary. However, the system has been designed with these standards in mind, and is hence compatible with both. Once the standards reach a final version, it could be beneficial for ChaNNSe to become compliant with these standards, potentially enabling it to interoperate with other standards-compliant systems.

7.6 Concluding Remarks

As computing in service-oriented architectures increases in popularity, more and more services will depend on other services. We believe that quality of service will be an important factor in selecting service providers, and that it should be possible for clients and service providers to come to an agreement over the QoS that they will receive, especially when the service is being paid for. We also believe that when multiple clients have similar interests, they should be able to benefit from their common interest, in the same way that group buy schemes are often organised between interest groups. This is especially so when it also benefits the service provider by not imposing as much load as serving each client separately. The ideas presented in this thesis go some way to progressing this field in this respect.

Bibliography

L. Amgoud and N. Maudet. Strategical considerations for argumentative agents (preliminary report). In *Proceedings of the 9th International Workshop on Non-Monotonic Reasoning (NMR 2002): Special Session on Argument, Dialogue and Decision*, pages 399–407, 2002.

L. Amgoud, N. Maudet, and S. Parsons. Modelling dialogues using argumentation. In *Proceedings of the 4th Conference on Multi-Agent Systems*, pages 31–38, Boston, MA, 2000.

A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu. Web services agreement specification (WS-Agreement), Feburary 2004.

P. Anthony. *Bidding Agents for Multiple Heterogeneous Online Auctions.* PhD thesis, School of Electronics & Computer Science, University of Southampton, 2003.

R. Axelrod. *The Evolution of Cooperation.* Basic Books, New York, 1984.

G. Banavar, T. D. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *Proceedings of the International Conference on Distributed Computing Systems 1999*, pages 262–271, Austin, TX, 1999a.

G. Banavar, T. D. Chandra, R. E. Strom, and D. C. Sturman. A case for message oriented middleware. In *Proceedings of the 13th International Symposium on Distributed Computing*, volume 1693, pages 1–18, London, UK, 1999b. Springer-Verlag.

M. Barbuceanu and W-K. Lo. A multi-attribute utility theoretic negotiation architecture for electronic commerce. In *Proceedings of the fourth international conference on Autonomous agents*, pages 239–246, Barcelona, Spain, 2000. ACM Press.

C. Bartolini, C. Preist, and N. R. Jennings. Architecting for reuse: A software framework for automated negotiation. In *3rd International Workshop on Agent-Oriented Software Engineering*, pages 87–98, Bologna, Italy, 2002.

C. Bartolini, C. Preist, and N. R. Jennings. *Software Engineering for Multi-Agent Systems III: Research Issues and Practical Applications*, volume 3390, chapter A Software

Framework for Automated Negotiation, pages 213–235. Springer-Verlag GmbH, January 2005.

R. Bilorusets, D. Box, L. F. Cabrera, D. Davis, D. Ferguson, C. Ferris, T. Freund, M-A. Hondo, J. Ibbotson, L. Jin, C. Kaler, D. Langworthy, A. Lewis, R. Limprecht, S. Lucco, D. Mullen, A. Nadalin, M. Nottingham, D. Orchard, J. Roots, S. Samdarshi, J. Shewchuk, and T. Storey. Web Services Reliable Messaging Protocol (WS-ReliableMessaging). msdn.microsoft.com/ws/2005/02/ws-reliablemessaging/, February 2005.

K. P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53, December 1993.

K. P. Birman. *Building Secure and Reliable Network Applications.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1 edition, 1996.

A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, 2(1):39–59, February 1984.

D. Box, L. F. Cabrera, C. Critchley, F. Crubera, D. Ferguson, A. Geller, S. Graham, D. Hull, G. Kakivaya, A. Lewis, B. Lovering, M. Mihic, P. Niblett, D. Orchard, J. Saiyed, S. Samdarshi, J. Schlimmer, I. Sedukhin, J. Shewchuk, B. Smith, S. Weerawarana, and D. Wortendyke. Web Services Eventing (WS-Eventing), August 2004.

S. Burbeck. The tao of e-business. www-128.ibm.com/developerworks/webservices/library/ws-tao/, October 2000.

A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 219–227, Portland, Oregon, July 2000.

E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. www.w3.org/TR/wsdl, March 2001.

M. Colan. Service-oriented architecture expands the vision of web services, part 1. www-128.ibm.com/developerworks/webservices/library/ws-soaintro.html, April 2004.

Comb-e-chem Project. Comb-e-chem website. http://www.combechem.org/, 2003.

K. Czajkowski, D. F. Ferguson, I. Foster, J. Frey, S. Graham, I. Sedukhin, D. Snelling, S. Tuecke, and W. Vambenepe. The WS-Resource framework. www.globus.org/wsrf/specs/ws-wsrf.pdf, May 2004.

R. K. Dash, D. Parkes, and N. R Jennings. Computational mechanism design : A call to arms. *IEEE Intelligent Systems*, 18(6):40–47, 2003.

V. Deora, J. Shao, G. Shercliff, P. J. Stockreisser, W. A. Gray, and N. J. Fiddian. Incorporating qos specifications in service discovery. In *Proceedings of the Second International Web Services Quality Workshop (WQW 2004)*, Brisbane, Australia, November 2004. Springer-Verlag.

A. Donnellan, J. Parker, G. Lyzenda, R. Granat, G. Fox, M. Pierce, J. Rundle, D. McLeod, L. Grant, and T. Tullis. The quakesim project: Numerical simulations for active tectonic processes. In *Proceedings of the 2004 Earth Science Technology Conference*, Palo Alto, 2004.

EBI. Swiss-prot website. http://www.ebi.ac.uk/swissprot/, 2003.

P. Th. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2):114–131, June 2003.

P. Faratin. *Automated Service Negotiation Between Autonomous Computational Agents.* PhD thesis, University of London, Queen Mary College, Department of Electronic Engineering, 2000.

P. Faratin, C. Sierra, and N. R. Jennings. Negotiation decision functions for autonomous agents. *International Journal of Robotics and Autonomous Systems*, 24(3–4):159–182, 1997.

P. Faratin, C. Sierra, and N. R. Jennings. Using similarity criteria to make negotiation trade-offs. In *Internation Conference on Multiagent Systems (ICMAS-2000)*, pages 119–126, Boston, MA, 2000.

P. Faratin, C. Sierra, N. R. Jennings, and P. Buckle. Designing responsive and deliberative automated negotiators. In *Proc. AAAI Workshop on Negotiation: Settling Conflicts and Identifying Opportunities*, pages 12–18, Orlando, FL, 1999a.

P. Faratin, C. Sierra, T. J. Norman, P. O'Brien, and J. L. Alty. Designing flexible automated negotiators: Concessions, trade-off and issue changes. Technical Report RR-99-03, Institut d'Investigacio en Intelligencia Artificial, 1999b.

FIPA. FIPA 97 part 2 version 2.0: Agent communication language specification. http://www.fipa.org/specs/fipa00003/, October 1998.

I. Foster. What is the grid? a three point checklist. *GRIDToday*, 1(6), July 2002.

I. Foster and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organisations. *International Journal of Supercomputer Applications*, 15(3):200–222, 2001.

G. Fox and S. Pallickara. JMS compliance in the Narada event brokering system. In *Proceedings of the 2002 International Conference on Internet Computing (IC-02)*, volume 2, pages 391–397, 2002.

G. Fox, S. Pallickara, M. Pierce, and H. Gadgil. Building messaging substrates for web and grid applications. *(To appear) in the Special Issue on* Scientific Applications of Grid Computing *in the Philosophical Transactions of the Royal Society of London 2005*, 2005.

D. Friedman. *The Double Auction Market Institution: A Survey*, pages 3–25. Addison-Wesley, Reading, MA, 1993.

D. Friedman and J. Rust, editors. *The Double Auction Market: Institutions, theories and evidence.* Addison-Wesley, Reading, MA, 1993.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing. Addison-Wesley, Boston, MA, 1st edition, 1995.

S. Graham, P. Niblett, D. Chappell, Sonic Software, Amy Lewis, Nataraj Nagaratnam, JayParikh, Sanjay Patil, Shivajee Samdarshi, Igor Sedukhin andDavid Snelling, Steve Tuecke, William Vambenepe, and Bill Weihl. Publish-subscribe notification for web services. Technical report, Akamai Technologies, Computer Associates International, Fujitsu Laboratories of Europe, Globus, Hewlett-Packard, IBM, SAP AG, Sonic Software and TIBCO Software, March 2004.

R. Granat. *Regularized Deterministic Annealing EM for Hidden Markov Models.* PhD thesis, University of California, 2004.

J. N. Gray. *Operating Systems: An Advanced Course*, chapter Notes on Database Operating Systems, page 393. Springer-Verlag, 1978.

GriPhyN Project. Grid physics network website. www.griphyn.org, 2005.

Paul Groth, Simon Miles, Weijian Fang, Sylvia C. Wong, Klaus-Peter Zauner, and Luc Moreau. Recording and using provenance in a protein compressibility experiment. In *Proceedings of the 14th IEEE International Symposium on High Performance Distributed Computing (HPDC'05)*, July 2005.

T. R. Gruber. A translation approach to portable ontologies. *Knowledge Acquisition*, 5 (2):199–220, 1993.

R. H. Guttman and P. Maes. Cooperative vs. competitive multi-agent negotiations in retail electronic commerce. In *Second International Workshop on Cooperative Information Agents (CIA '98)*, volume 1435 of *Lecture Notes in Computer Science*, pages 135–147, Paris, France, 1998.

M. Hapner, R. Burridge, R. Sharma, J. Fialli, and K. Stout. Java message service version 1.1. Technical report, Sun Microsystems, April 2002.

M. He and N. R. Jennings. Designing a successful trading agent: A fuzzy set approach. *IEEE Transactions on Fuzzy Systems*, 12(3):389–410, June 2004.

M. Horstmann and M. Kirtland. DCOM architecture. Technical report, Microsoft Corporation, 1997.

Y. Huang and H. Garcia-Molina. Publish/subscribe in a mobile enviroment. In *Proceedings of the 2nd ACM international workshop on Data engineering for wireless and mobile access*, pages 27–34, Santa Barbara, CA, 2001. ACM Press.

M. Humphrey, G. Wasson, M. Morgan, and N. Beekwilder. An early evaluation of wsrf and ws-notification via wsrf.net. In *Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing (GRID'04)*, pages 172–181, Pittsburgh, PA, November 2004. IEEE Computer Society.

N. R. Jennings. On agent-based software engineering. *Artificial Intelligence*, 177(2): 277–296, 2000.

N. R. Jennings, P. Faratin, A. R. Lomuscio, S. Parsons, C. Sierra, and M. Wooldridge. Automated negotiation: Prospects, methods and challenges. *International Journal of Group Decision and Negotiation*, 10(2):199–215, 2001.

N. R. Jennings, S. Parsons, P. Noriega, and C. Sierra. On argumentation-based negotiation. In *Proceedings of the International Workshop on Multi-Agent Systems*, Boston, USA, 1998.

N. R. Jennings, S. Parsons, C. Sierra, and P. Faratin. Automated negotiation. In *5th International Conference on thePractical Application of Intelligent Agents and Multi-Agent Systems*, pages 23–30, Manchester, UK, 2000.

N. C. Karunatillake and N. R. Jennings. Is is worth arguing? In *Proceedings of First International Workshop on Argumentation in Multi-Agent Systems (ArgMAS 2004)*, pages 32–77, Columbia University, NY, USA, 2004.

P. Klemperer. Auction theory: A guide to the literature. *Journal of Economic Surveys*, 13(3):227–286, July 1999.

S. Kraus, K. Sycara, and A. Evenchik. Reaching agreements through argumentation: a logical model and implementation. *Artificial Intelligence*, 104(1–2):1–69, September 1998.

A. Krishna, S. Miles, L. Moreau, and M. Luck. Semantic distributed messaging middleware. In *Proceedings of the ECAI 2004 Workshop on Semantic Intelligent Middleware for the Web and the Grid (SIM'04)*, Valencia, Spain, August 2004. http://CEUR-WS.org/Vol-111/.

A. Krishna, V Tan, R. Lawley, S. Miles, and L. Moreau. The myGrid notification service. In *Proceedings of the UK OST e-Science second All Hands Meeting 2003 (AHM'03)*, pages 475–482, Nottingham, UK, September 2003. ISBN ISBN - 1-904425-11-9.

Y Labrou, T. Finin, and Y. Peng. Agent communication languages: the current landscape. *Intelligent Systems*, 14(2):45–52, 1999.

K. Larson and T. Sandholm. An alternating offers bargaining model for computationally limited agents. In *Proceedings of the firstinternational joint conference on Autonomous agents and multiagent systems*, pages 135–142, 2002. ISBN 1-58113-480-0.

R. Lawley, K. Decker, M. Luck, T. Payne, and L. Moreau. Automated negotiation for grid notification services. In *9th International Euro-Par Conference (EUROPAR 03)*, Klagenfurt, Austria, 2003a.

R. Lawley, M. Luck, K. Decker, T. Payne, and L. Moreau. Automated negotiation between publishers and consumers of grid notifications. *Parallel Processing Letters*, 13(4):537–548, December 2003b.

R. Lawley, M. Luck, and L. Moreau. Chained negotiation for distributed notification services. In *Proceedings of the UK OST e-Science second All Hands Meeting 2004 (AHM'04)*, Nottingham, UK, September 2004.

R. Lawley, M. Luck, and L. Moreau. Modelling & simulating chained negotiation to enable sharing of notifications. In *Proceedings of IEEE/WIC/ACM Int Conf on Web Intelligence*, Compiegne, France, 2005.

P. Li and B. Ravindran. Proactive qos negotiation in asynchronous real-time distributed systems. *Journal of Systems and Software*, 73(1):75–88, September 2004.

H. Ludwig, A. Dan, and R. Kearney. Cremona: An architecture and library for creation and monitoring of ws-agreements. In M. Aiello, M. Aoyama, F. Curbera, and M. Papazoglou, editors, *Proceedings of the 2nd International Conference on Service Oriented Computing (ICSOC 2004)*, pages 65–74, New York, NY, 2004. ACM Press.

G. Malkin and A. Harkin. TFTP option extension (RFC 1782). Technical report, Network Working Group, 1995.

P. McBurney and S. Parsons. Locutions for argumentation in agent interaction protocols. In R. M. van Eijk, M-P. Huget, and F. Dignum, editors, *Agent Communication. Revised Proceedings of the International Workshop on Agent Communication (AC2004)*, number 3396 in Lecture Notes in Artificial Intelligence, pages 209–225, New York, NY, July 2004.

Sun Microsystems. Java remote method invocation specification. http://java.sun.com/j2se/1.4.2/docs/guide/rmi/index.html, 2003.

S. Miles, J. Papay, T. Payne, M. Luck, and L. Moreau. Towards a protocol for the attachment of metadata to service descriptions and its use in semantic discovery. *Scientific Programming*, 14:201–211, 2005.

Simon Miles, Juri Papay, Vijay Dialani, Michael Luck, Keith Decker, Terry Payne, and Luc Moreau. Personalised grid service discovery. *IEE Proceedings Software: Special Issue on Performance Engineering*, 150(4):252–256, August 2003. ISSN 1462-5970.

L. Moreau, M. Luck, S. Miles, J. Papay, K. Decker, and T. Payne. *Methodologies and Software Engineering for Agent Systems*, chapter Agents and the Grid: Service Discovery. Kluwer, 2004.

L. Moreau, S. Miles, C. Goble, M. Greenwood, V. Dialani, M. Addis, N. Alpdemir, R. Cawley, D. De Roure, J. Ferris, R. Gaizauskas, K. Glover, C. Greenhalgh, P. Li, X. Liu, P. Lord, M. Luck, D. Marvin, T. Oinn, N. Paton, S. Pettifer, M. V. Radenkovic, A. Roberts, A. Robinson, T. Rodden, M. Senger, N. Sharman, R. Stevens, B. Warboys, A. Wipat, and C. Wroe. On the Use of Agents in a BioInformatics Grid. In Sangsan Lee, Satoshi Sekguchi, Satoshi Matsuoka, and Mitsuhisa Sato, editors, *Proceedings of the Third IEEE/ACM CCGRID'2003 Workshop on Agent Based Cluster and Grid Computing*, pages 653–661, Tokyo, Japan, May 2003. ISBN 0-7695-1919-9.

G. Mühl. *Large-Scale Content-Based Publish/Subscribe Systems.* PhD thesis, Darmstadt University of Technology, September 2002.

myGrid Project. myGrid website. http://www.mygrid.org.uk/, 2003.

T. Norman, A. Preece, S. Chalmers, N. R. Jennings, M. Luck, V. Dang, T. Nguyen, V. Deora, J. Shao, W. Gray, and N. Fiddian. Conoise: Agent-based formation of virtual organisations. *Knowledge-Based Systems*, 17:103–111, 2004.

OASIS. Web services reliable messaging tc ws-reliability. http://www.oasis-open.org/committees/download.php/5856/WS-Reliability-2004-03-10.pdf, January 2004a.

OASIS. Web services topics 1.2 (ws-topics), July 2004b.

OASIS. Web Services Base Notification 1.3 (WS-BaseNotification), July 2005a.

OASIS. Web Services Brokered Notification 1.3 (WS-BrokeredNotification), July 2005b.

Object Management Group. Notification service specification. www.omg.org, aug 2002.

Object Management Group. Event service specification. www.omg.org, 2004.

T. Oinn. Change events and propagation in mygrid. Technical report, European Bioinformatics Institute, 2002. http://www.ebi.ac.uk/ tmo/change_notification.pdf.

M. J. Osborne and A. Rubinstein. *Bargaining and Markets.* Academic Press, Inc, San Diego, California, 1990.

M. J. Osborne and A. Rubinstein. *A Course In Game Theory.* MIT Press, Cambridge, MA, 1994.

S. Pallickara and G. Fox. An approach to high performance distributed web brokering. *ACM Ubiquity*, 2(38), November 2001.

S. Pallickara and G. Fox. NaradaBrokering: A middleware framework and architecture for enabling durable peer-to-peer grids. In *Proceedings of ACM/IFIP/USENIX International Middleware Conference Middleware-2003*, number 2672, pages 41–61. Springer, 2003.

S. Pallickara and G. Fox. On the matching of events in distributed brokering systems. In *Proceedings of IEEE ITCC Conference on Information Technology*, volume 2, 2004a.

S. Pallickara and G. Fox. A scheme for reliable delivery of events in distributed middleware systems. In *Proceedings of the IEEE International Conference on Autonomic Computing*, pages 328–329, New York, NY, 2004b.

S. Pallickara and G. Fox. An analysis of notification related specifications for web/grid applications. In *Proceedings of the IEEE ITCC Conference on Information Technology 2005*, pages 762–763, 2005.

S. Pallickara, G. Fox, and S. Lee. An analysis of reliable delivery specifications for web services. In *Proceedings of the IEEE ITCC Conference on Information Technology 2005*, pages 360–365, 2005.

S. Parsons and P. McBurney. Argumentation-based dialogues for agent coordination. *Group Decision and Negotiation*, 12(5):415–439, 2003.

S. Parsons, C. Sierra, and N. R. Jennings. Agents that reason and negotiate by arguing. *Journal of Logic and Computation*, 8(3):261–292, 1998.

J. Patel, W. T. L. Teacy, N. R Jennings, and M. Luck. A probabilistic trust model for handling inaccurate reputation sources. In P. Herrmann, V. Issarny, and S. Shiu, editors, *Proceedings of Third International Conference on Trust Management*, pages 193–209, Rocquencourt, France, 2005.

M. Perry, C. Delporte, F. Demi, A. Ghosh, and M. Luong. *MQSeries Publish/Subscribe Applications.* IBM Redbooks, 2001.

J. B. Postel. RFC 821 — Simple Mail Transfer Protocol. www.faqs.org/rfc/rfc821.html, August 1982.

I. Rahwan, S. D. Ramchurn, N. R. Jennings, Peter McBurney, Simon Parsons, and Liz Sonenberg. Argumentation-based negotiation. *The Knowledge Engineering Review*, 18(4):343–375, 2004.

H. Raiffa. *The Art and Science of Negotiation.* Harvard University Press, 1982.

J. S. Rosenschein and G. Zlotkin. *Rules of Encounter: Designing Conventions for Automated Negotiation among Computers.* MIT Press, Cambridge, MA, 1994.

K. Rothermel, G. Dermler, and W. Fiederer. Qos negotiation and resource reservation for distributed multimedia applications. In *Proceedings of the 1997 International Conference on Multimedia Computing and Systems (ICMCS '97)*, pages 319–335, Ottawa, Canada, 1997. IEEE Computer Socirty.

B. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *In Proceedings of AUUG97*, pages 243–255, Canberra, Australia, September 1997.

B. Segall, D. Arnold, J. Boot, M. Henderson, and T. Phelps. Content based routing with elvin4. In *Proceedings AUUG2K*, Canberra, Australia, June 2000.

ServoGRID Project. Solid earth research virtual observatory grid website. www.servogrid.org, 2005.

C. Sierra, P. Faratin, and N. R. Jennings. A service-oriented negotiation model between autonomous agents. In *Proceedings of the 8th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW-97)*, pages 17–35, Ronneby, Sweden, 1997a.

C. Sierra, N. R. Jennings, P. Noriega, and S. Parsons. A framework for argumentation-based negotiation. In *Intelligent Agents IV: 4thInternational Workshop on Agent Theories Architectures and Languages*, volume 1365 of *LNAI*, pages 177–192. Springer, 1997b.

H. A. Simon. A behavioral model of rational choice. *The Quarterly Journal of Economics*, 69(1):99–118, February 1955.

R. Stevens, K. Glover, C. Greenhalgh, C. Jennings, S. Pearce, P. Li, M. Radenkovic, and A. Wipat. Performing in silico experiments on the grid: A users perspective. In *Proceedings of the UK OST e-Science second All Hands Meeting 2003 (AHM'03)*, pages 43–50, Nottingham, UK, 2003.

M. T. Tu, F. Griffel, M. Merz, and W. Lamersdorf. A plug-in architecture providing dynamic negotiation capabilities for mobile agents. In K. Rothermel and F. Hohl, editors, *Proceedings of the 2nd International Workshop on Mobile Agents*, volume 1477 of *Lecture Notes in Computer Science*, pages 222–236, Stuttgart, Germany, 1998. Springer-Verlag.

UDDI. Universal description discovery and integration (UDDI). http://www.uddi.org/, 2001.

A. Uyar, S. Pallickara, and Fox. G. Towards an architecture for audio video conferencing in distributed brokering systems. In *Proceedings of the 2003 International Conference on Communications in Computing*, pages 17–23, 2003.

J. von Neumann and O. Morgenstern. *The Theory of Games and Economic Behaviour.* Princeton University Press, 1953.

Web Services Architecture Working Group. Web services architecture. www.w3.org/TR/ws-arch/, February 2004.

Web Services Reliable Messaging TC. WS-Reliability 1.1. Technical report, OASIS, 2004.

E. Wolfstetter. Auctions - an introduction. Technical Report 1994-13, Humboldt Universitaet Berlin, 1994. http://ideas.repec.org/p/wop/humbsf/1994-13.html.

M. Wooldridge. Agent-based software engineering. *IEE Proceedings Software Engineering*, 144(1):26–37, 1997.

World Wide Web Consortium. Soap version 1.2 part 1: Messaging framework. http://www.w3.org/TR/soap12-part1/, June 2003.

G. Zacharia, A. Moukas, and P. Maes. Collaborative reputation mechanisms in electronic marketplaces. In *Proceedings of the 32nd Hawaii Internation Conference on System Sciences*, Maui, Hawaii, January 1999.