NOMAD:

Towards an Architecture for Mobility in Large Scale Distributed Systems

Kristian Paul Bubendorfer

A thesis,

submitted to the Victoria University of Wellington
in fulfilment of the requirements for the degree of
Doctor of Philosophy in Computer Science.

Victoria University of Wellington

December 21, 2001

To Andrea

Abstract

Applets, proxies and increasingly complex web applications point toward the Internet becoming a globally distributed system. Currently these services are provided in an *ad hoc* fashion at fixed locations, which users locate and access manually. In addition, services are typically provided with little or no regard to the user's location or quality of service requirements.

The Nomad architecture developed in this thesis directly addresses these problems and forms the basis of a global infrastructure for mobile object-based distributed system. Applications within the system negotiate and pay for execution resources in order to distribute themselves to meet quality of service requirements. High level support is provided to ease the task of programming mobile applications, reducing the time between prototype and deployment.

The major contributions of this thesis are the negotiation and location services. Applications are composed of multiple mobile objects and negotiate for resources using the Vickrey auction protocol. The Nomad negotiation architecture addresses all the shortcomings of the chosen auction protocol, ensuring that within our domain, the protocol makes optimal allocations of execution resources. Supporting the resulting distributed applications is the Nomad location service. The location service architecture transfers responsibility for monitoring the location of an application's mobile objects to the application itself. This has the significant advantage of widely distributing the load incurred when tracking highly mobile objects. These application held fragments of the location service are then integrated into a unified global location architecture.

Acknowledgements

I have an enormous debt of gratitude to the following people. My long time supervisor John Hine, who not only guided me through my MSc, but once again risked his sanity by supervising this PhD. Richard Hayton, specifically for providing help and debugging by email of Flexinet, and the rest of the ANSA team for making Flexinet available. Ken Moody and Jean Bacon for their hospitality during my four week visit to the Computing Laboratory, Cambridge. John Haywood for his statistical advice. My wife Andrea, for reading and improving my drafts, even when she'd rather be feeding ducks at the park. My thanks also to those unnamed, but essential people in the Computer Science Group, who have at times been harassed in the corridors or cornered in their offices.

Lastly, I'd like to thank my thesis examiners; Ken Moody, James Noble and Andry Rakotonirainy, for their insightful questions and comments which have undoubtedly strengthened my thesis.

Let euerie eye negotiate for it selfe, And trust no Agent.

- **1599** W. Shakespeare. Much Ado II. i. 185

Contents

1 Introduction **1**

1.1 Goals and Strategies . . . 2

1.2 Contributions . . . 4

1.3 Thesis Organisation . . . 4

2 Related Work **5**

2.1 Mobility in Distributed Operating Systems . . . 6

2.2 Economic Resource Management in Computer Systems . . . 7

2.2.1 Critique . . . 10

2.3 Specific Systems . . . 11

2.3.1 Emerald . . . 12

2.3.2 Globe . . . 13

2.3.3 Legion . . . 14

2.3.4 D'Agents . . . 15

2.4 Summary . . . 16

3 Nomad - A Middleware Architecture **18**

3.1 Economic Resource Management Model . . . 20

3.2 Nomad Design Overview . . . 21

3.3 Application Architecture . . . 22

3.3.1 Structural Application Requirements . . . 22

3.3.2 Making and Receiving Payments . . . 23

3.3.3 Mobile Clusters . . . 23

3.3.4 Distribution . . . 24

3.3.5 Transparency . . . 25

3.4 Nomad . . . 26

3.4.1 Contracts . . . 26

3.4.2 The Negotiation Service 26
3.4.3 The Location Service 27
3.5 Depot Architecture 28
3.5.1 Policy Model 29
3.5.2 Persistence 31
3.6 Summary 32

4 Naming and Location **33**
4.1 Naming 34
4.1.1 Required URN Functional Capabilities 34
4.1.2 URN Syntax 35
4.1.3 Resolution Requirements 36
4.2 Existing Location Services 37
4.2.1 The DNS 38
4.2.2 Globe 39
4.2.3 Flexinet 39
4.3 The Nomad Location Service 40
4.3.1 Exploiting Application Locality 41
4.3.2 Design Overview 42
4.3.3 Creating an Application 44
4.3.4 Registering a Service 45
4.3.5 Obtaining a Service 47
4.3.6 Moving an Object 48
4.3.7 Rebinding an Object 48
4.3.8 Replication of the Location Tables 51
4.3.9 Access Rights 52
4.3.10 Internal Architecture of Location Table Locator and YellowPages . . 53
4.4 Fault Model 53
4.5 Summary 54

5 Negotiation **55**
5.1 Relationships Between Participants 56
5.1.1 Potential Impacts of Hostility 57
5.2 Metrics of Negotiation 58
5.3 General Negotiation Mechanisms 59

5.3.1 Informational Negotiation 60
5.4 Competitive Negotiation 60
5.4.1 Bipartite Negotiation 61
5.4.2 Multipartite Negotiation and Auction Protocols 63
5.4.3 Valuation 64
5.4.4 Auction Equivalence 65
5.4.5 Dominant Strategies 65
5.4.6 The Benefit of Truthful Bidding 66
5.4.7 Protocol Limitations 67
5.5 Summary 69

6 Representation of Resources **71**
6.1 Representations 71
6.1.1 Utility Function 71
6.1.2 The Combinatorial Allocation Problem 72
6.1.3 Prećis 75
6.2 Resource Description Graph 75
6.2.1 The Graph 76
6.2.2 The RDG Combinatorial Allocation Problem 78
6.2.3 Graph Composition 80
6.2.4 Versatility of Description Graphs 80
6.2.5 Textual RDG Representation 81
6.2.6 Edge expressions 82
6.3 Pattern Matching 84
6.4 Summary 84

7 A Negotiation Mechanism for Nomad **85**
7.1 The Market 86
7.2 The Negotiation 88
7.2.1 Registering an Auction 88
7.2.2 Catalogue Distribution 89
7.2.3 Bidding 90
7.2.4 Closing the Auction 91
7.2.5 Redeeming the Contract 92
7.3 Entirety Bids 92

7.4 Incorporating Bipartite Negotiations 93
7.5 Limitations Revisited 94
7.6 The Global Marketplace Structure 95
7.6.1 Categories 96
7.6.2 Catalogue Distribution 97
7.6.3 Ordering 97
7.7 Auction properties 98
7.8 Marketplace Economics 98
7.9 Fault Model 99
7.10 Summary 100

8 Experimental Prototype **101**
8.1 Prototype 101
8.1.1 Implementation 102
8.2 Location Service 102
8.3 Negotiation System 103
8.3.1 Tribbles 104
8.3.2 Experimental Platform 105
8.3.3 Random Valuation 106
8.3.4 Uniform Cost Model 107
8.3.5 Nonuniform Cost Model 108
8.3.6 Tribble Constraint 110
8.4 Whiteboard 111
8.4.1 The Client-Server Model 112
8.4.2 The Distributed Application Model 115
8.4.3 Précis 122

9 Conclusions **123**
9.1 Review 124
9.1.1 The Negotiation Service 125
9.1.2 The Location Service 127
9.1.3 The Prototype 127
9.2 Future Work 128
9.2.1 Experimental Verification 128
9.2.2 Additional Implementation 128

9.2.3 Additional Research 129
9.3 Summary 129

A Interfaces **131**
A.1 Location Service 131
A.2 Market 131
A.2.1 Clients 131
A.2.2 Market 132

B Whiteboard Implementation **133**
B.1 Whiteboard Client Implementation 133
B.2 Whiteboard Service Implementation 134

References **137**

Chapter 1

Introduction

Organisations with global clientele are facing the problem of integrating a multitude of applications and data-sources, while delivering services to wherever a customer or client exists. The Internet is providing the fabric on which it is possible to build globally distributed applications, however this still relies on the organisation providing the resources from which to serve its clients.

This collection of internetworked computers represents a considerable and poorly utilised computational resource. SETI@home [55] was the first large-scale use of the Internet as a massive computational resource utilising donated processor time from idle internetworked computers.

Nomad (Negotiated Object Mobility Access and Delivery) is a middleware platform that provides a distributed system infrastructure for the deployment of applications composed of lightweight mobile objects. The provision of this high-level infrastructure shortens the time to prototype, reduces application complexity, and improves interoperability through a consistent world view (standardisation). Mobility allows applications to reorganise their structure in response to user demands, to move closer to fixed resources, and to improve reliability through access to facilities such as replication and persistent storage. The Nomad architecture supports such applications by providing a mechanism to negotiate with resource providers called *Depots* for runtime resources where and when they are required.

This thesis describes the design, and proof-of-concept implementation of the Nomad architecture. The first part of this introduction is a statement of the goals and strategies

underlying the Nomad architecture. This is followed with an overview of the layout of the thesis and the contributions of this work.

1.1 Goals and Strategies

Much thought has gone into determining the requirements of future middleware architectures. In a recent paper, Geihs [33] outlined the challenges ahead for middleware, and described a set of architectural goals which are summarised below in point 1. The intention behind Nomad is also to create a scalable open system, where any application or host may participate — these goals are captured in the second and third points.

1. Support mobility as a first class paradigm for the future:
   (a) provide distribution transparencies,
   (b) but permit contextual awareness (selective transparencies),
   (c) allow customisable middleware architecture,
   (d) support adaptive applications, and
   (e) meet QoS specifications.
2. Provide an open system:
   (a) globally allocate computational and other resources,
   (b) support and take advantage of heterogeneity,
   (c) control the behaviour of each mobile application with respect to the local goals and policies of the administrative domain being visited,
   (d) protect applications from malicious hosts, and
   (e) encourage the deployment and building of infrastructure.
3. Ensure scalability:
   (a) design scalable global infrastructure,
   (b) minimise overheads to both the system and its applications, and
   (c) automate management where possible.

With the goals from points 2 and 3 firmly in mind, the question is, how to achieve them. In the design of Nomad, the conviction is that these structural goals can be achieved with the following strategies:

1. The best way to build long-term viable global infrastructure and to encourage the provision of resources is to charge for their use [17].

2. Local selfish optimisation by consumers and producers in an economic system is a good basis for global resource allocation [66, 31, 66, 93, 109].

3. High level infrastructure and mobility with transparencies enable rapid implementation and deployment of applications.

Management in a mobile system is needed by both applications — to meet client demands, and by the Depots on which they run, to ensure fair and reliable allocation of resources. Figure 1.1 shows a high-level view of the Nomad architecture, where human managers set policy, and then the entities themselves manage the day to day operation of the system or application. Human intervention should only be required due to unforeseen circumstances like component failure, malformed policies, or routine policy refinement.

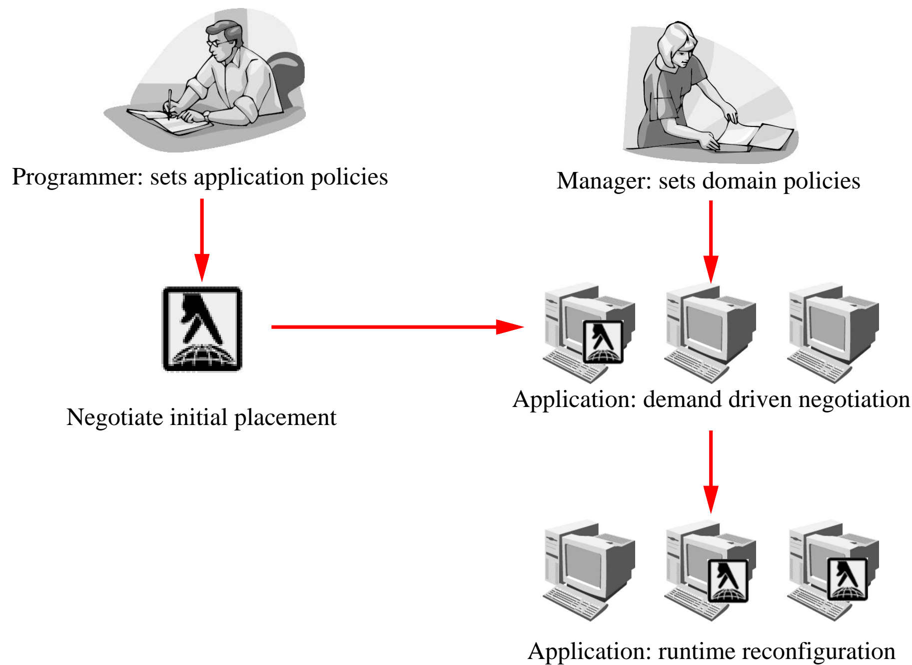


Figure 1.1: *A high-level overview of the NOMAD architecture.*

1.2 Contributions

This thesis draws together disparate areas of research in a novel approach to distributed system architecture, and documents the resulting design and development of the system along with the implementation of a prototype. The Nomad service layer, incorporating the location and negotiation services, provides a unifying view of the system for both applications and Depots. The Nomad middleware architecture captures all salient points of mobile system design, and contributes significantly to the understanding of how to design and implement large scale computer systems.

1.3 Thesis Organisation

This thesis is organised as follows. Chapter 2 reviews the literature from two distinct fields, economic resource allocation and mobile systems. The second part of the chapter reviews the recent systems in which these areas have begun to converge.

Chapter 3 introduces the Nomad architecture, providing a broad overview of the entire middleware system, and the design of the mobile application and Depot architectures.

Chapters 4 through 7 form the main body of this thesis. Chapter 4 documents the Nomad location service, a global scalable solution to the problem of tracking mobile objects.

The issues underlying resource negotiation are investigated in chapter 5 and the criteria required for a negotiation mechanism are investigated. The ability for an application to describe its resource requirements to its potential hosts is critical if an application is to find the resources it requires in a heterogenous system. This problem is addressed in chapter 6 with resource description graphs.

The Nomad *Marketplace* is developed in chapter 7, utilising the analysis from chapter 5 and the resource description graphs chapter 6, to provide a mechanism for economic computational resource allocation.

Aspects of the Nomad architecture are verified experimentally in chapter 8 and chapter 9 concludes this thesis, reviewing the contributions and areas requiring additional research.

Chapter 2

Related Work

Both mobility and economic resource allocation models have their origins in the late 1970s and early 1980s, and this chapter seeks to provide a context which brings together these two diverse bodies of work. Figure 2.1 shows the time-line of previous systems, from Medusa [75] and ContractNet [97] in 1980, through to the latest research systems that incorporate economic resource models in large scale distributed systems.

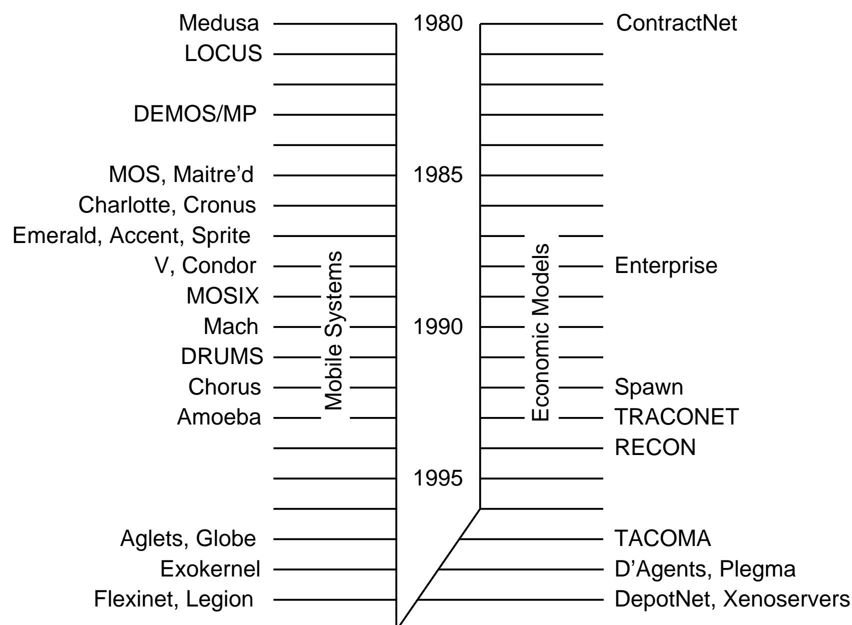


Figure 2.1: *Time-line, converging of mobile systems and economic resource allocation.*

This chapter will start by discussing early systems with mobility in distributed operating systems, then economic resource management in computer systems. The systems from

figure 2.1, D'Agents, Globe, Legion and XenoServers represent recent research and are discussed in detail in section 2.3. Flexinet [40] is a special case, as the Nomad prototype is built directly on top of it, and section 4.2.3 provides an overview. DepotNet [17] represents the first publication of what has since become the Nomad architecture.

2.1 Mobility in Distributed Operating Systems

Medusa [75] was a distributed operating system for the Cm* multi-computer which performed initial task placement over its individual processors. Despite the low power of the processors and minimal kernel, Medusa provided an early implementation of location transparency. LOCUS [82], one year later, was a UNIX based distributed operating system, with location independent naming and transparent system calls (via the file system). A later extension [96] to the system in 1983 added a limited ability for processes to migrate between homogeneous hosts — this mechanism was limited however, and there was no policy basis in the LOCUS system. DEMOS/MP [83], from 1983, was an extension of the original DEMOS message passing operating system and featured a process migration mechanism. Location was by the use of forwarding pointers, rather than via a global structure as with the LOCUS file system. In later research DEMOS/MP acquired a load aware process manager and migration policy [68].

The MOS [7] from 1985 was the first system architecture designed from the start for full process migration. Built on a network of UNIX computers — the main focus (and contribution) was on the distribution of load information and a range of associated sophisticated scheduling policies. Another system from the same year was Maitre d' [10], which did not have migration, but initial task placement. The point of interest is that there was a degree of mutual selection between the source and destination machines, where the destination machine may opt out of receiving the task.

Up until this time all of the systems involved heavyweight tasks or processes, and were oriented towards overall system performance. In 1986 the Cronus [94] object oriented distributed system appeared. All communication was through invocation on objects via message passing. Cronus objects are somewhat heavyweight, examples of which include processes and directories. Each object is directly controlled by a manager process on an arbitrary host. The resources of the system are represented by classes, each of which is

associated with a manager process for that class. The manager process for each class is resident on each machine which has that resource. Support for location transparency permits invocation to be independent of the client or object location. Invocation on a migrated object is obtained by contacting the object manager responsible for the object (determined from the UID), which in turn forwards the message. The following year, Emerald [51] featured fine-grained object mobility, and was the first distributed object programming language to adopt application directed object mobility. This was an important development, and Emerald is detailed in section 2.3.1.

Subsequent years continued with the refinement of migration and initial placement mechanisms, with lazy migration in Accent [86], user transparency in Sprite [29], removal of residual dependencies in Charlotte [2] and reduction of migration times by pre-copying in V [22]. Different distribution policies, information collection and system network architectures were variously studied, with ownership of single user stations in Condor [63], PVM[1] in MOSIX [6], migrating load information servers in DRUMS [13], a processor pool model in Amoeba [103], variable policies in Mach [12], and distributed shared virtual memory with objects in COOL [1, 61].

In contrast Aglets [32, 60], concentrated on very light weight mobile objects (agents), as in Emerald, but on top of Java. The goal behind Aglets was migration in its own right — that is, a mobile form of Applet. Some effort was made to enable the safe execution of untrusted Aglets, however no distributed system infrastructure was provided. Exokernels [52] allowed applications to *opt-out* of system resource control. The effect was one of application level control through separating resource protection from management, and safely allowed the applications to exert custom control over the resources that they required. Considerable performance gains were achieved with this architecture, reflecting that system management of resources is at best a compromise. This is also an underlying principle used in the design of Nomad.

2.2 Economic Resource Management in Computer Systems

Economics has long inspired researchers as a means of allocating resources. Economics is a well founded discipline with much analysis and research based on everyday human

[1] Parallel Virtual Machine.

systems that is directly applicable to computer systems. The discussion of the economic resource management systems in this section is more detailed than the previous section, as the economic approach to system resource management is not well documented in the existing computer science literature.

The earliest published computation market was the 1968 *futures* market [101], used to manually allocate blocks of computer time on a departmental PDP-1. Users bid, in advance, for computer time by marking their initials and price for their desired period on a paper scheduling sheet. This is an open-ascending auction, and later bidders can preempt earlier bids until the auction closes, after which a recorded telephone message was used to inform users of successful bids. Priority was controlled by allocating different amounts of currency to various users and projects. The scheme was found to give very high machine utilisation, and ease of management due to the simple prioritisation mechanism.

The next step was in 1980 with the contractNet protocol [97], developed to perform resource negotiations between distributed problem solvers (DPS). This is a cooperative system, rather than competitive, so no currency is exchanged. Each DPS has a set of distinct *knowledge sources*, which in combination with other DPSs are sufficient to solve the problem. When a DPS (the *manager*) has a task which requires execution, it announces (broadcasts) a description of the task, and DPSs which are idle (potential *contractors*) bid on those announcements which are compatible with their set of knowledge sources. The bids are evaluated by the manager and the most suitable contractor (best fit) is selected. The contractor is responsible for executing the task, and the manager is responsible for monitoring the contractor's execution and processing the results. Contractors may further subdivide a task and subcontract to other DPSs — for which the contractor will in turn, act as a manager. This is the first system to use a negotiation process involving mutual selection by managers and contractors.

In 1988 Miller and Drexler [31, 69] developed a computation market for the efficient auctioning of processor time to non-concurrent processes on a single processor computer system. This was also probably the first system which considered the application of the various forms of auction protocol to computer resource management. As the auction was for CPU time on a single processor, a form of priority was needed to prevent starvation. The solution was to use a variation on the Vickrey [107] auction which linearly increased all bids over time, so that later bids would need to be higher to compete with earlier bids

(*escalation*).

At the same time, Enterprise [66] was developed for distributed computing systems. Enterprise performed initial placement of tasks to processors, using a protocol based on that of contractNet [97], that is, no auction and no currency. In place of currency, Enterprise used an estimate of priority based on the expected processing time. Cheating by misstating or erroneous processing time estimates was policed by automatically aborting processes that significantly overran their estimate. Kurose and Simha [59] in 1989 investigated an economic iterative auction model for the distributed allocation of files, and demonstrated the attractive properties of feasibility, monotonicity, and fast convergence.

1992 saw the first implementation of a market-based economic resource management system for distributed computing systems with Spawn [109]. Spawn was the first full attempt to create a distributed computational economy, and utilises the bid escalation from [31, 69]. As with the previous systems, the only resource negotiated for was processor time, and the currency in the system was allocated to users by human system administrators. Users were prioritised by allocating different amounts of currency as in [101]. Each computer in Spawn executed two processes, an *auctioneer* and a *resource manager*. The resource managers each contained a small list of auctioneers on neighbouring computers which supplied updates of pricing and availability. The auctioneer continuously accepted bids on the next available slice of time. Spawn was limited to initial placement, and so on expiry of an existing processor slice, the choice was either to abort the application or to let it continue execution. The approach used was to leave the decision to the application, that is, by offering the application first-refusal basis on the next processor slice at the current price. When an application wished to *spawn* a new child, it sent a request to the local resource manager which then matched the applications requirements against those available on the monitored auctions, and bid on its behalf.

TRACONET [90] from 1993 extended the original contractNet protocol to include marginal costing, giving a more market-like resource allocation mechanism. Apart from the calculation of the costing and bid prices, the protocol is the same — and was applied to solve the distributed vehicle routing problem, where each dispatch centre was represented by an intelligent agent. The next step in 1994 was RECON [30], again a distributed problem solver, built on top of WALRAS [110] which dynamically adjusted and extended the market structure itself to include markets for solutions to new tasks and subtasks.

In 1997, a position paper [50] on TACOMA detailed the inclusion of electronic cash in the system to enable payment for services. The mechanism was subtle in its implementation, but not actually part of the underlying architecture since TACOMA had no economic resource model or negotiation system. Nonetheless, it illustrates that electronic payments within a large scale distributed system are feasible.

Plegma [53] is a distributed problem solving architecture built for medical imaging systems, and utilising a market-based (contractNet [97]) resource management mechanism. Each agent (essentially a contractNet knowledge source) is responsible for a set of local resources and has a set of problem solving abilities. Plegma consists of three main entities; clients, providers and the marketplace (PLEGMA infrastructure). Charging for execution was based on the bids received when assigning tasks to agents, and the marketplace was explored via simulation.

Xenoservers [112, 88] also promote the ideal of a computational economy with mobile objects. The planned system has extensive parallels with the Nomad architecture, with Xenoservers and Nomad Depots forming loci of resource and computation, the Xenoserver Trader and the Nomad Market for Xenoserver and Depot discovery, and resource negotiation.

2.2.1 Critique

The systems in this section do not all present economic models for resource allocation in distributed systems. Rather it is an overview of a diverse body of work all of which reflect in some way on the design of Nomad.

ContractNet, TRACONET, RECON and Plegma are distributed problem solvers, using markets and agents to either represent, or to compute solutions. ContractNet in particular was the first negotiation system to utilise mutual selection, as does Nomad, while RECON reconfigures and extends the markets to include markets for solutions to new tasks. This is similar to the inclusion of categories, and the dynamic configuration of the Nomad Marketplace.

Miller and Drexler developed what was the first computation resource market utilising a modified Vickrey auction protocol. However, the modification which escalated the apparent price of early bids (to prevent starvation) reintroduced counterspeculation and

consequently non-optimal allocations in a non cooperative system. This concept is fully explained in section 5.4.6. TACOMA is a practical demonstration that it is feasible to include payment systems in large scale distributed systems architectures.

Enterprise utilises an expected process lifetime as the basis of its *currency*. This is useful as later work by Harchol-Balter and Downey [37] demonstrate that a very simple memoryless estimate of the expected lifetime can be computed. In particular, the probability that a process of age T will use an additional T is $\approx \frac{1}{2}$. However, Enterprise is far from a viable solution as there is no provision for computing the market price, that is, clients cannot make tradeoffs between fast and expensive or slow and cheap contractors.

Spawn was the first implementation of a market-based economic resource management system. However, the design has a number of flaws, including use of the escalation modification to the Vickrey protocol and no advertising mechanism outside a fixed set of machines, a set which is probably too small to give optimal allocations. There is also no appreciation of combinatorial allocation — only one resource is negotiated, which seems strange for a system touted as heterogeneous, as there is no guarantee that any particular machine is capable of executing any particular application. The final problem is that the negotiations are not a result of mutual selection - the host machine performs negotiation on the application's behalf. Auctions are also for free resources rather than application requirements. In these respects Spawn differs significantly from the Nomad approach, but is also the most advanced economic resource negotiation system in this section and was a considerable leap forward.

Additionally, none of these systems dealt with any resources other than CPU time. There is no representation of other resources that exist in distributed systems, such as bandwidth. Any solutions, in the context of the problems addressed in this thesis, need to be able to represent all types of resources.

2.3 Specific Systems

This section presents a survey of large scale distributed object, negotiation and location systems. While not all the systems included in this section claim to operate in or are designed for the same domain as Nomad, all contain at least part of an applicable solution for a truly global distributed system. The systems with strong relevance, shared goals, or

that are still current projects are detailed in this section.

2.3.1 Emerald

The Emerald programming language [46, 51] was the first effort to demonstrate the utility of fine grained objects in the construction of distributed applications and systems. Designed as a language as well as an infrastructure for distributed environments, Emerald included features for locating objects and extending exception handling to cope with partial failure of the distributed system. Of particular significance however, is that mobility was application directed, with language primitives.

System Overview

Emerald extended the single object model to distributed systems, with objects being the only entities represented in the language. Everything in Emerald is an object — with objects being the only entities represented in the language. Each Emerald object is immutable and has: a unique name, a set of fields, a set of methods, an optional process and a location. The only mechanism for communication is invocation, which is location independent. Location information is maintained by the host on which the object was first created, leaving lifelong residual dependencies. Objects are garbage collected when there are no longer any references to them.

Mobility

Distribution within Emerald is achieved with language primitives to control the placement and movement of objects. *Locate* determines where an object resides, *Move* moves an object to another location, *Fix and Unfix* control whether an object may move, and *Refix* which unfixes, moves and fixes an object. In addition to these primitives, Emerald also offers a parameter passing mode *call-by-move* during invocation. A call-by-move parameter is *moved* to the destination host during invocation, and then either returned after the call completes or remains on the destination host depending on the specified mode. These primitives ensure Emerald recognition as the first distributed system to utilise application directed mobility as a language paradigm.

The granularity of mobility in Emerald is very small - based on the unit of the object. Efficiency is improved by freely copying small objects such as Integers and other immutable objects. All objects in Emerald are 'friendly' and hosts homogeneous, and as such there is no security, access or resource control.

2.3.2 Globe

Globe [3, 4, 39, 41, 44, 43, 42, 105, 106] is an active research project based on a wide area distributed system architecture designed to provide the infrastructure on which to develop large scale wide area applications.

The basic premise behind this architecture is the *distributed shared object* which appears to be simultaneously located on multiple machines. Through this distributed object's interface services such as: communication protocols; replication strategies and the distribution of state are made available. The interface gives full distribution transparency.

System Overview

In Globe each object offers at least one interface, comprising of a set of methods by which that object is manipulated. The objects are *passive*, and may be physically distributed over a set of machines as shown in figure 2.2. Each distributed object has a *contact-point*, which is the first point of contact for all binding clients. The Nomad architecture also uses an application structure similar to the contact-point.

Globe local objects are the components from which distributed shared objects are constructed. A local object resides in only one address space, and implements the distributed object's interface. As Globe distributes objects over a heterogeneous network, the local implementation of the distributed object interface will likely have a different implementation. How the local object is actually implemented depends on a number of pragmatics including the platform architecture and the object replication policy. Figure 2.2 depicts the local objects in each of the address spaces that the distributed shared objects spans. Each Globe object maintains consistency of its own distributed state.

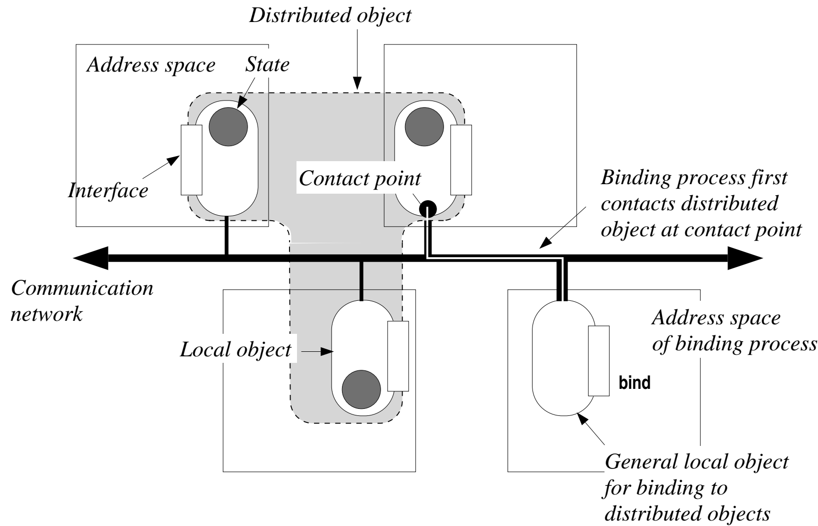


Figure 2.2: *Illustration of a Globe distributed object, figure reproduced from [105].*

Mobility

The Globe distributed shared object is a logical composite of its local objects across the resident address spaces. The various local objects themselves perform all distribution, replication and movement of the object's state in such a way that all clients are provided with a consistent view of the distributed object's state.

There is no mechanism for resource control, again all entities are treated as friendly. The Globe operating system is required by all clients interacting with distributed shared objects, and while this insulates clients from the inherent complexities and reduces programmer load, it in turn increases the size and complexity of the system itself. A major contribution of Globe was its location service, which is presented separately in section 4.2.2.

2.3.3 Legion

The Legion [35, 35, 36, 21] software architecture is designed to abstract a large collection of heterogeneous machines as a single virtual machine, providing a simple uniform interface.

System Overview

Legion is a middleware system of distributed objects that sits between Legion applications and the host operating system. All entities within Legion e.g. application objects

and hosts, are represented by independent active objects which communicate via remote method invocation.

Resource Control

Resources in Legion are abstracted in two independent objects namely the host and vault objects. Hosts represent computation and vaults represent persistent storage. As Legion sits above a standard unmodified operating system, no control is expressed over very low level resources - this is the job of the host operating system. As various hosts may have different policies, the implementation of the classes are host dependant. Currently there are no accounting policies to discourage *free loading*. Also, the hosts decide the execution priorities of objects, without consideration of application requirements.

Legion Critique

Legion is a single meta-machine abstraction, on which users execute possibly distributed applications. One of the goals of Legion is to support heterogeneity, taking advantages of features such as vector processors or massively parallel machines. However, the placement and later migration of objects is the responsibility of the object class. There is no mechanism for an application to find the resources available in the heterogeneous environment and thereby exploit them. This means that application behaviour and system information must be encoded statically in the application's object classes. Each object class is also responsible for the location of objects of its type, which is similar to an extent in object oriented databases [19]. Obviously this level of control by a class limits its later reuse to the same set of heterogeneous machines and the applications for which they were constructed.

For a general purpose large scale distributed system, it appears that the level of abstraction of the Legion meta-machine is too high. Legion also lacks resource discovery and therefore the ability for the application to tailor its execution to the availability of runtime resources.

2.3.4 D'Agents

D'Agents [16, 24, 15, 56] is an active research project — with similar goals to those of Nomad. Mobile agents inhabit a heterogeneous system and move across the network to

collocate data and computation.

System Overview

D'Agents is a mobile agent system, where programs migrate under their own control from machine to machine within a heterogeneous network. Each host in the system provides interpreters to execute mobile objects written in Java, TCL and Scheme. D'Agents provides security systems to provide protection to hosts from malicious agents.

Resource Control

An economic system is used to control the excessive consumption of resources, especially as the system spans multiple administrative domains. Resources are represented individually by resource managers on each machine, and mobile agents use a non-convertible form of electronic cash to exchange for resources. Each mobile agent is created with a finite supply of cash and dies on exhaustion of these funds. There is no mechanism for a mobile agent to acquire additional funds.

Resources are sold using a centralised Vickrey auction in [14], where each individual resource manager auctions its capacity to mobile agents. In a more recent implementation [16] the auction protocol is a repeated first-price sealed bid auction.

D'Agents Critique

D'Agents negotiations are seller initiated, and buyers may therefore only buy resources offered by the hosts. D'Agents has no mechanism for agents to describe their needs to hosts, and no mechanisms to combine conditional arbitrary sets of resources from separate auctions, see chapter 6.

2.4 Summary

This chapter has traced the evolution of mobility and the use of economic resource allocation models. After 1996 these two formerly disjoint fields began to converge with the application of economic resource allocation to the problems of mobile distributed systems. The exception to this trend was Spawn, which explored this model several years earlier.

As recently as 1999, the economic distributed mobile object systems did not deal with the allocation of resources other than CPU. All resources require consideration to achieve reasonable utilisation of heterogeneous systems and permit applications to take advantage of unique features. The solution to this problem requires general mechanisms to discover and negotiate for all resources required by an application at run time.

Chapter 3

Nomad - A Middleware Architecture

Nomad (Negotiated Object Mobility Access and Delivery) is designed to support applications consisting of dynamic mobile objects. These objects are supported by a network of *Depots* that provide resources ranging from CPU cycles and persistent storage, to communication and consistency protocol in exchange for payment.

Figure 3.1 gives an overview of the major components within the Nomad architecture. All execution within Nomad takes place on virtual hosts (vHosts) which are managed along with other Depot resources by each Depot's local manager. These Depots, their managers, vHosts and resident clusters (see section 3.3.3) of client, application and nomad objects are the physical entities within Nomad and are drawn in a solid line.

The remaining components from figure 3.1 (Client, Application, Marketplace and Location Services) are shown in dashed lines and represent functionality, rather than describe a set of physical resources or the distribution of an execution. For example, the application represents a service that is supplied to a client and is shown in dashed lines. However, the component clusters of this application, the Contact Object (CO), Service Object (SO), and Location Table (LT) physically reside in vHosts on Depot A and Depot B — the LT cluster is also shown as functionally part of the location service (on the left of the figure).

An application controls its degree of distribution by placing and replicating its objects to achieve goals such as reduced latency to clients, redundancy and concurrent processing. The application negotiates with the Depots via the Nomad Marketplace for the resources

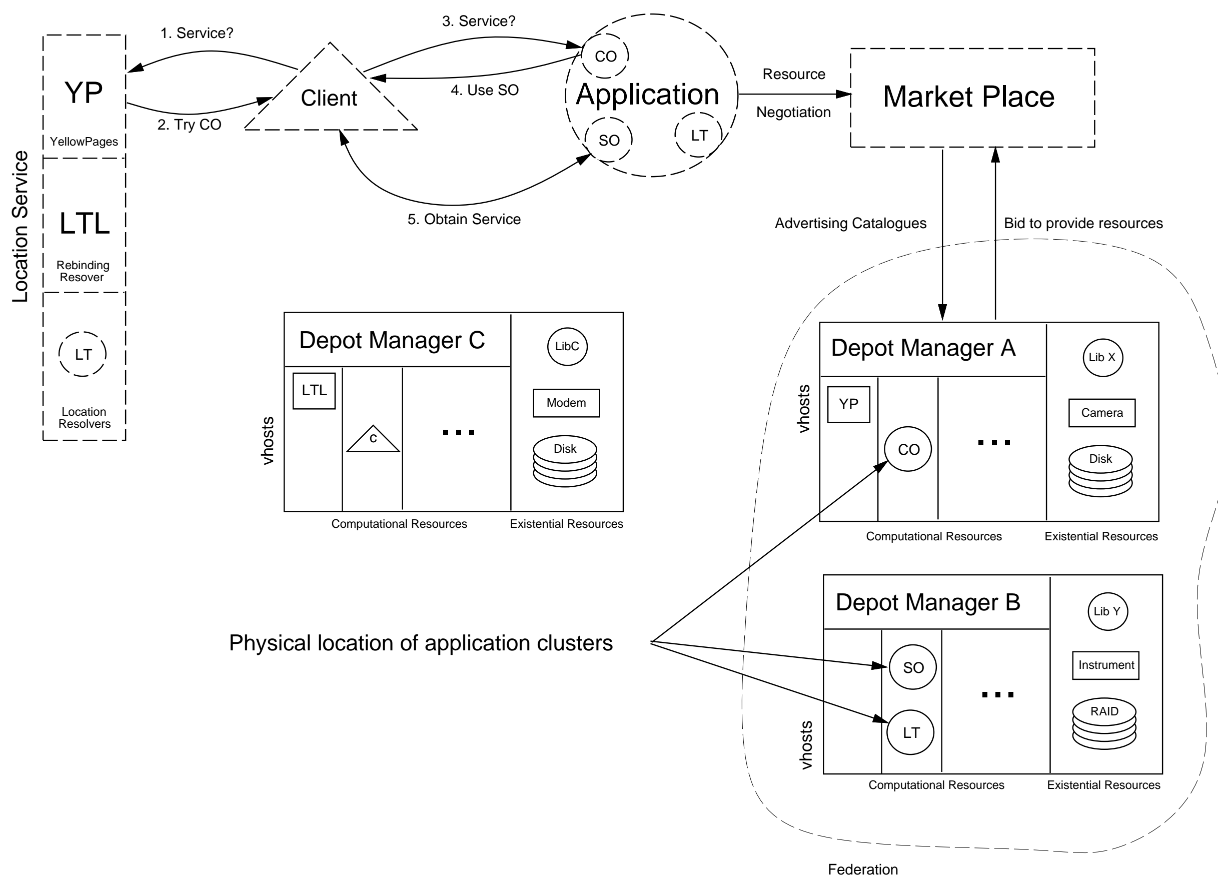


Figure 3.1: *High level overview of the components within the Nomad Architecture.*

that it requires. For their part, Depots use policies to price their resources and to organise such things as the balance of work amongst a Federation of Depots, or the quality of service that an individual Depot offers. A federation of Depots reflects common ownership, or administration of the Depots within that federation.

A client wishing to obtain a service contacts the central service directory, the YellowPages, with the well-known service name. The YellowPages associates this name with a specific application and returns a reference to an application contact object (see section 3.3.1). The client then negotiates with the contact object for the service which is ultimately provided via a service object (again see section 3.3.1). The Location Table Locator (LTL) keeps track of the Location Tables, which are in turn used to track the location of the rest of an application's mobile objects.

The remainder of this chapter presents the economic resource management model used in Nomad, followed by a more detailed bottom-up description of the various components.

These include: the application architecture, the global Nomad service architecture, and the Depot architecture.

3.1 Economic Resource Management Model

Traditionally resource management is undertaken by the operating system. In this role, it has the conflicting goals of optimising the resource allocation for the entire system (*globally*) and for each application (*locally*). This is difficult as the operating system is neither aware of, nor privy to, the needs or goals of each application. A solution to this problem is to separate the local optimisation from the global optimisation, that is, the application must negotiate for the resources it requires from the system on which it wishes to execute [100].

This is similar to the problem of allocating finite resources over an independent population — as faced by all human societies. For example, a farmer wishes to get the best price for his wheat, while the baker wishes to pay less. The solutions to this and related problems have been refined by many years of human competition and forms the basis of economics [69]. This well founded and understood technique of *resource pricing* has long been applied to the problem of allocating finite computational resources [101].

The economic model is also powerful enough to provide a mechanism, with which to solve a number of other problems relating to the behaviour of entities within the system. One such problem in open mobile systems is how to prevent mobile objects from maliciously (or otherwise) wasting or over-using resources on a host they are visiting. This is especially a concern when the host and object have different administrative domains or ownership. Rather than designing a system that predicts the entire range of possible actions an application might take, and then permits or forbids these actions depending on the probable impact on the system, financial incentives [31] can be used to control the behaviour of entities within the system. An example of this is the denial of service attack which, if the instigator is charged for the actual resources consumed in the attack, becomes infeasible.

The incentives in the construction of the economic system are designed such that it is in an entity's *best interest* to behave in a desirable way [93]. Infractions cost real currency, and as a consequence badly behaving entities suffer a real financial loss and are unable to utilise additional resources once their currency is expended. This is not a complete

panacea, as the system must still ensure that the entities within the system are protected from other forms of interference, and that the payment and negotiation systems are secure. This is a considerably easier problem than that involving the non-economic approach.

3.2 Nomad Design Overview

The Nomad middleware architecture consists of three separate layers, the Depot layer, the unifying Nomad Service layer and the Application layer. These layers are illustrated in figure 3.2.

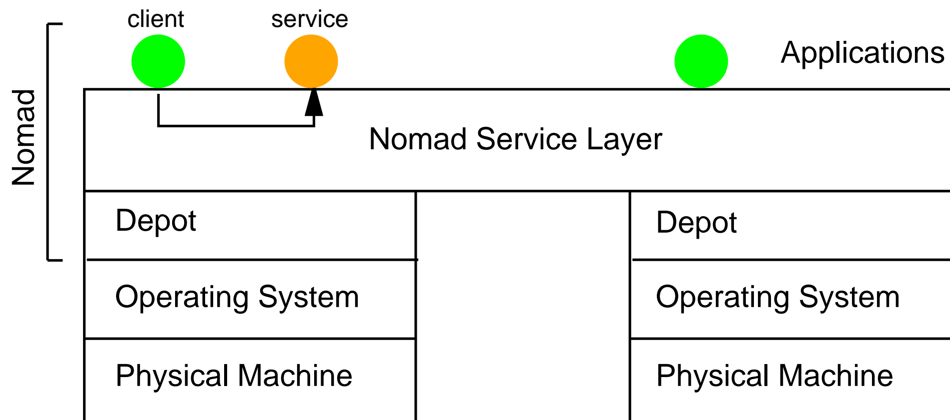


Figure 3.2: *The Nomad middleware architectural layers.*

An application is a cohesive collection of objects and is structured as a set of mobile clusters. Each application is responsible for: initiating negotiation for resources, directing the distribution of its clusters, handling unrecoverable failures, and maintaining part of the location service. Applications provide services to clients, and may themselves be clients of other applications.

The Nomad layer services execute on the network of Depots and are responsible for providing a consistent global infrastructure to applications and Depots. This infrastructure includes resource negotiation, location services and a payment service. Resource negotiation includes mechanisms for performing negotiation, ensuring the integrity of negotiations and policing contract compliance. The location service unifies the location service components, maintained by applications, into one cohesive global structure.

A Depot is comprised of a set of physical resources and an associated manager. Each Depot is responsible for: supporting computation and communication, implementing security and fault detection, providing storage and mobility, and responding to negotiation requests. Depots manage themselves via policies which are specified by a human manager, and may include coordination with other Depots in the same administrative Domain.

3.3 Application Architecture

An application in Nomad is required to implement a two-level contact-service object hierarchy, which is described in section 3.3.1, and is composed of a set of mobile clusters, as detailed in section 3.3.3. The application experiences transparencies, see section 3.3.5, while executing within Nomad, and negotiates with Depots via the Nomad negotiation service to distribute these clusters in response to client needs, as described in section 3.3.4.

3.3.1 Structural Application Requirements

Application services to clients are delivered by a set of service objects, but the first point of contact between an application and its clients is the well known application contact object. Globe has a similar idea, with its contact-point [105], but this is very much more limited in functionality.

The contact object negotiates[1] with clients, and determines how the required service is to be delivered. This client-service negotiation is different from the resource negotiation provided by the Nomad negotiation service layer as discussed in section 3.4.1. A contact object for a service may, for instance, offer different levels of service, gold, silver and bronze — for which it will charge accordingly.

Once negotiation has been successfully completed, the contact object returns a service object interface to the client. Using this interface, the client then transparently interacts with the service object regardless of the service object's actual location.

This distinction between contact objects and service objects is required to address the dynamic configuration of an application and its ability to meet Quality of Service (QoS)

[1] It is expected that the majority of negotiations will simply involve stating the one time fee, followed by acceptance from the client.

requirements. Both service and contact objects are potentially replicated within the application, and the use of formal contact objects allows the application to distribute the client load over its service objects, or indeed redeploy or create additional service objects on demand.

3.3.2 Making and Receiving Payments

The distribution of currency amongst the objects of an application is application dependent, however one potential mechanism is to utilise an internal currency distribution tree. Spawn [109] uses this mechanism and feeds funds down through the application and its subtasks in discrete units, which are further divided at each branch. A reservoir of funds is retained by the root to cope with short-term cash-flow problems.

When application income is collected by the contact object, this mechanism can be extended to feed income collected at the leaves back up the tree. This is an ideal structure for distributing funds within an application.

3.3.3 Mobile Clusters

A cluster rather than an object is the Nomad unit of mobility. An application composed of objects must be able to manipulate all of its objects. In most object oriented languages this implies being able to locate and bind to quite small objects such as Integers. For distribution, this is generally too fine-grained, and the smallest meaningful unit for mobility will consist of one or more objects which provide some encapsulated functionality. Such an encapsulated grouping is a *cluster*.

A cluster also provides an address space that allows references within the cluster to be handled by the implementation language's mechanisms. A new object is created within a cluster and remains within that cluster for its lifetime. (A clone of the object may be created in a different cluster.) A cluster may or may not contain threads of its own.

Objects in different clusters are in different address spaces. Nomad provides mobile remote method invocation (MRMI) so that an object in one cluster may invoke a method on an object in a different cluster independent of either of the clusters' physical locations. MRMI also rebinds low level network connections when an object moves.

3.3.4 Distribution

An application controls its own distribution, and is expected to meet client QoS demands. Moving closer to the client or a fixed resource is a well recognised technique to improve performance [73, 11, 74].

An application could be structured in a variety of ways, it could maintain a pool of service objects, of which the closest is used to service a client, or it could create replicas on demand. These structures require the distribution process to occur at different times, *preemptively* and *on demand.*

Preemptive Distribution

The application maintains a pool of distributed service objects from which it attempts to satisfy new and existing clients. Once the application determines there is insufficient future capacity in a subset of the service objects, it negotiates for and creates additional service objects. If the existing application structure cannot cope with a sudden increase in demand, then the application must either refuse new clients or offer a lower level of service.

The problem the application faces is working out future client load and subsequent application requirements. Preemptive distribution is sensible for frequently used lightweight services, where the individual setup cost (in time or currency) exceeds the value of the service.

On Demand Distribution

With on demand distribution, the application creates additional application capacity in response to a new client, or to maintain existing clients.

The negotiations for new resources to deploy additional service objects and other application structures are likely to result in some delay before the client can obtain the service. A small delay is acceptable if the service is a long term investment, such as streamed video.

3.3.5 Transparency

There are nine transparencies identified in the ISO/IEC standard [49]: access, location, failure, replication, security, migration, relocation, persistence and transaction. Many distributed systems aim to present total transparency to the application; that is, neither an application nor its objects are aware of the location of any objects, including themselves.

Total transparency however limits the ability of an application to organise itself in a way that is consistent with its goals [108]. The system is not privy to all of the information about an application[2], nor could it be, and as such cannot make the right choice for every application in every circumstance:

> *"While it is critical that the invocation is location independent, or that distribution be transparent with respect to invocation, it is not necessary that an object's location be invisible. Many applications may choose to ignore distribution, but others may wish to benefit from location dependence."* — *Jul et al.[51]*

This position is reflected in Nomad, where an application can opt out of even the default transparencies. Migration, relocation, replication, persistence, transaction, and failure are application directed or resolved, while security, access and location[3] are normally transparent. To the client of an application, all interactions except unrecoverable failures are transparent.

All distribution within Nomad is Application driven. The complexity that would normally be imposed on the application programmer is in the most part mitigated by the inherited `AuthMobileObject` class. For example, migration involves no more than a mobile object invoking the `Migrate` method on itself:

$$\text{Migrate}(\textit{dest},\ \textit{authority});$$

The authority is a certificate representing permission to migrate onto the destination host.

[2]Including the undocumented intentions of the programmer.

[3]In the RM-ODP terminology communication transparency is provided by a combination of access and location transparencies. Thus in this case location transparency only applies to communications rather than placement as might be initially thought.

This certificate is the result of prior negotiation for the right to access resources on the destination.

3.4 Nomad

The Nomad services provide global infrastructure. In particular, provide a location service and an impartial negotiation mechanism. The Nomad level services run on the Depots and each Depot shares part of the overhead. The Nomad infrastructure is independent of any federation structure.

3.4.1 Contracts

There are two contracts formed in Nomad, the contract resulting from a resource negotiation between a Depot and application, and the contract between a client and a service. The Nomad resource negotiation service is concerned with the first type of contract, the second is application specific and is supported by the Nomad application architecture.

- **Application to Depot**: The Nomad negotiation service is used to negotiate a contract that contains an agreed set of resources and QoS specifications. How the Depot will satisfy these QoS requirements is an active and complementary research field [34, 20, 48, 84].

- **Client to Application**: This contract is negotiated through the application's contact object using application specific QoS specifications. The application may redistribute its service objects to meet the client's QoS requirements which may require negotiation with Depots for additional resources. This contract also includes the means by which the client will pay for the application's services.

3.4.2 The Negotiation Service

Markets in particular are an ideal foundation for resource management when used with a resource price model:

> *Market price systems constitute a well understood class of mechanisms that under certain conditions provide effective decentralisation of decision making with minimal communication overhead — Wellman [110].*

Indeed, as chapter 2 has illustrated, such economic systems have been used for the allocation of computational resources since 1968 [101]. Markets are particularly appropriate in situations where software is spread across a distributed system, serving different clients and pursuing different goals [69].

The properties of this distributed decision making, with minimal overhead, well understood theory and the abstraction of resources as currency ensure that the economic resource model is the ideal choice for use in Nomad. This permits the Nomad system to charge for the use of resources, prevent unbounded resource attacks as with denial of service [112], and represent priority. A Market provides a forum in which buyers and sellers meet (resource discovery) and negotiate to perform optimal allocations with minimal overhead, mutual selection and decentralised decision making between buyers and sellers.

Thus negotiation in Nomad is performed in a market, operating on the principles of the Vickrey auction as analysed in chapter 5. Applications requiring a contract construct a description of their requirements using resource description graphs from chapter 6 and negotiations take place using the Nomad *Marketplace* negotiation mechanism detailed in chapter 7.

3.4.3 The Location Service

The Nomad location service provides the means for any object within Nomad to locate and invoke methods on any other object. Chapter 4 presents a solution based on sharing the task of tracking mobile clusters between global location systems, and the local application to which those clusters belong. This approach ensures that the tracking of mobility is widely and fairly distributed, and that the application is able to customise its local contribution to meet its functional requirements. This division gives decentralised control, the ability to scale, and efficient use of system resources.

3.5 Depot Architecture

A Depot is a pool of vHosts, which provide computational resources and a consistent execution environment over heterogeneous hosts. Each vHost provides a secure execution environment to the clusters of a single application, along with transparent method invocation, fault detection and basic mechanisms supporting mobility and persistence, as shown in figure 3.3. All references external to a cluster are accessed via stubs resident in the cluster and therefore in its namespace. Objects of the same application, on the same vHost, but in different clusters also require stubs. This is because there is no guarantee that, during their interaction, one cluster will not move to another vHost or Depot.

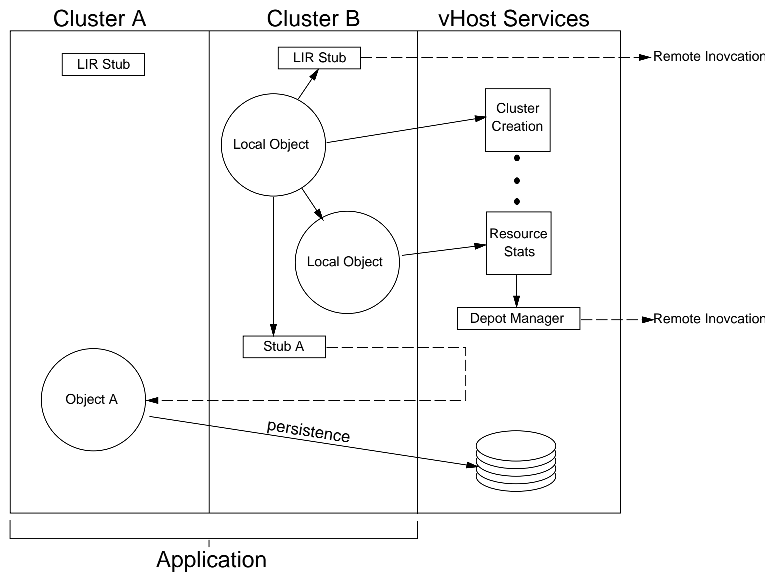


Figure 3.3: *One vHost with its one client application.*

The pool of vHosts is managed by a Depot manager, which itself executes on a vHost. This Depot structure is illustrated in figure 3.4. The Depot vHost contains a Resource Manager (RM) which manages and allocates the vHosts in the pool, a Statistics Collector (SC) which collects and distributes machine resource statistics for all the managed vHosts, and the local interface repository (LIR). Resource management and negotiation are implemented via policies, as described in section 3.5.1.

This figure also shows the second vHost which is reserved for the exclusive use of the Nomad level services, such as the location service and the Market. The client application (C) is shown performing a remote method invocation on the LIR in the Depot Manager

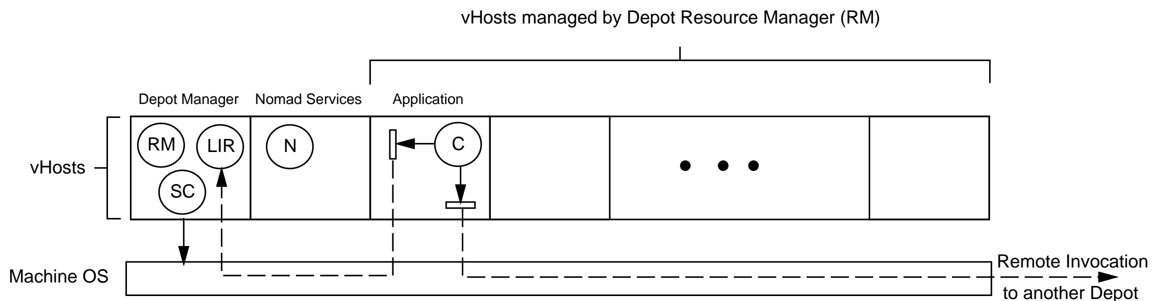


Figure 3.4: *The Depot is a pool of vHosts, Depot Manager and resident Nomad services.*

vHost, and a second invocation on an object on some other Depot.

The LIR contains references to services such as the location service, Marketplace and resource manager which may be required by clients executing on the vHost. The LIR executes in its own cluster on the Depot Manager vHost, and an interface on it is available in each cluster in each vHost. The LIR is local to a Depot and cannot be accessed from outside the Depot.

3.5.1 Policy Model

Each Depot is controlled by management policy which dictates how the Depot will react in specific circumstances: how it will ensure levels of service, its allocation policy, its negotiation policy, and how it interacts with Depots in and outside its federation. The Depot architecture separates mechanism and policy. Policy can be applied through a number of mechanisms such as: finite state machines, policy languages or standard general purpose programming languages.

Some policy is general, while some is highly structured. PONDER [27] is a policy language which is able to neatly capture general policies, where there is little interdependence. For instance, the human manager may want the Depot to perform a differential backup of its persistent store whenever it has free cycles. These sorts of policies are naturally expressed in PONDER:

```
on lowLoad
  subject s=persistentStore
do s.differentialBackup()
```

PONDER does not naturally express policies where there are large numbers of variables

and interactions and this is typical of resource management. Since the resource policies are the most important subclass of Depot policies, there is a requirement for an additional specialised resource management policy structure. Resource management can be broken down into three policy subclasses:

$$\text{Resource Management Policies} \begin{cases} \text{Negotiation Policies,} \\ \text{Compliance Policies,} \\ \text{Scheduling Policies.} \end{cases}$$

Negotiation Policies establish which resource allocations are made, *Compliance Policies* determine how a Depot enforces compliance with the negotiated allocations, and *Scheduling Policies* direct the Depot on how to satisfy the negotiated allocations.

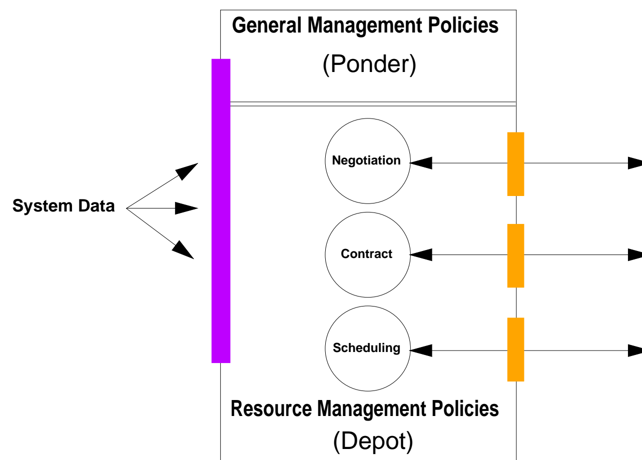


Figure 3.5: *The Policy Model within a Depot*

The Depot implements these policies in the policy structure shown in figure 3.5. Each of the interfaces, indicated by the rectangular blocks, hides the implementation of the policies from the system. Consider the following example of a compliance policy expressed in English:

> *If an application is currently using more resources than its negotiated peak rate, then providing it hasn't already used more resources than its total contracted amount (if any) and it hasn't done this before, then issue a warning to the application. However, if this is not the application's first transgression, or it has already used all of its allocated resources, then evict it.*

This is encoded into Depot policy with the following implementation in Java:

```
ResourceProfile profile = resources.usage(application);

if(contract.isNotComplying(profile)){
  if(contract.exhausted(profile)||contract.getOffences() > 1){
     manager.evict(application, new Reason(Reason.BREECH_OF_CONTRACT));
  }else{
     manager.warn(application, profile, new Reason(Reason.BREECH_OF_CONTRACT));
     contract.setOffences(contract.getOffences()+1);
  }
}
```

3.5.2 Persistence

There are two forms of persistence supported within a Depot. The first is normal *persistence*, the second is a weaker form termed *quiescence*. Both forms of persistence are negotiated with the Depot manager prior to arrival on the Depot, and the persistence mechanism is implemented by the individual vHosts. All persistent objects are also required to be mobile and implement either the `Persistent` or `Quiescent` interfaces.

A vHost hosting a persistent object will serialise and write to disk the entire mobile cluster of a persistent object, after each invocation on the persistent object, and before the result of the invocation is returned. If the object, vHost or Depot crash, the entire mobile cluster is deserialised from disk and restarted automatically. The exact basis for payment of this service needs additional consideration, it is however, likely to come at a premium.

Quiescence reflects the needs of a different type of application. Rather than deserialising and activating the cluster on a vHost, the serialised cluster is instead written directly to stable storage. The cluster remains quiescent on the Depot until it is needed again, at which time the cluster is activated and deserialised onto a local vHost by a `wakeup` method invocation on its `Quiescent` interface. The negotiated contract is intended to cover the resource requirements of the cluster once it is woken from quiescence, however there may not be sufficient Depot resources to reactivate the cluster immediately and activation is necessarily on a best-effort basis.

3.6 Summary

This chapter has presented the Nomad applications architecture, and details how the combination of the consistent local infrastructure provided by the Depot architecture, and the consistent global infrastructure provided by the Nomad service layer, combine to form a consistent and interlocking environment for highly distributed mobile applications.

The use of the resource price model and Markets is motivated, and the basis of the location service is highlighted. These latter Nomad services of Negotiation and Location are detailed in the following chapters (4 through 7) of this thesis.

Chapter 4

Naming and Location

Mobility experienced by a system comes in two forms: physical mobility, with mobile phones and laptops; and software mobility, with mobile objects and agents. To reference any entity in a distributed system, it must firstly be identifiable (by a name), and secondly be locatable. These functions of naming and location are often combined in systems without mobility or replication, and in these cases it is sufficient to encode the address information within the object name.

The most prominent example of this fusion of naming and location is the Uniform Resource Locator (URL) [9]. With mobile entities[1] however, the URL changes as the object moves. Without knowledge of the new URL, the object can no longer be accessed. In addition, replication is difficult, even when replicas remain at fixed addresses, as all replica addresses would need to be encoded in that URL.

The solution to this problem is to separate the name of an entity from its location, and then use a location service to resolve the name and return the present location of the entity. The specific goals of the Nomad location service are that:

- A client should be able to locate and bind to a target application for which it holds only a name,
- The relocation of an object should be supported in a manner that is transparent to the users of that object, and

[1] A physically mobile entity can be represented with a mobile software object, consequently, the term mobile object will be used to denote both.

- An application should be able to locate and bind to all of its internal objects.

In addition to these requirements, a successful location service will have the characteristics of:

- **Stability:** External and higher level[2] bindings should be stable, changing infrequently despite the dynamic nature of the application components.
- **Scalability:** The algorithms should scale with the number of objects.

This chapter will first discuss naming followed by the major part of the chapter which introduces the design of the Nomad location system and shows how this fits within the proposed IETF[3] uniform resource name (URN) naming scheme.

4.1 Naming

The first step is a separate human readable textual name. The URN [98, 99, 95] is a human readable name which is translated into a physical location (possibly a URL) by a URN resolver.

> *The purpose or function of a URN is to provide a globally unique, persistent identifier used for recognition, for access to characteristics of the resource or for access to the resource itself [98].*

4.1.1 Required URN Functional Capabilities

It is the functional capabilities required of the URNs as specified in RFC1737 [98] which make them suitable in a global infrastructure.

- Global scope: a URN is a name which does not imply a location and has the same meaning everywhere.
- Global uniqueness: the same URN will never be issued to two different resources.

[2]Hence the Nomad two level application architecture involving contact and service objects, see section 3.3.1.

[3]The Internet Engineering Task Force.

- Persistence: the URN may outlast the resource it identifies or any naming authority involved in the assigning of its name.
- Scalability: URNs can be assigned to any resource that might conceivably be available on the network for hundreds of years.
- Extensibility: any URN scheme must permit future extensions.
- Independence: it is solely the responsibility of the name issuing authority to determine the conditions under which it will issue a name.
- Resolution: a URN will not impede its resolution into a location — there must be some feasible mechanism to perform the resolution.
- Legacy support: the scheme must support existing naming schemes insofar that they satisfy the other functional requirements, i.e. ISBN, ISO etc.

The more of these requirements that a URN scheme meets, the more effective it will be.

4.1.2 URN Syntax

In addition to the qualities of persistence, location-independence, etc., URNs are intended to make it easy to map other namespaces (legacy support) into URN-space. Therefore the URN syntax must provide a means to encode character data in a form that can be sent using existing protocols and transcribed using most keyboards [70].

The URN has the following simple structure:

```
<URN> ::= "urn:"<NID>":"<NSS>
```

where:

- `"urn:"`

  identifies the string as a URN,

- `<NID>`

  is the namespace identifier, and

- `<NSS>`

  is the namespace specific string.

The NID distinguishes between different namespaces, or contexts, which may be administered by different authorities. As mentioned previously, such NIDs could include ISO, ISBN, etc. The NSS is only valid within its particular NID context, and has no predefined structure. As pointed out in [99], the significance of this syntax is in what is missing, as it is these omissions which make the URN persistent. That is, there is no communications protocol specified, no location information, no file system structure and no resource type information. As an example of a URN, a book may well have the name, `urn:ISBN:0-00-433245-8`, for which the information returned might well be a digital catalogue entry. A URN could also act as a textual representation of a telephone number, for example, `urn:phone:nz:wellington:bubendorfer:kris`, with the information returned being the telephone number.

4.1.3 Resolution Requirements

Having discussed the functional requirements of a URN scheme, we now need to look at the corresponding requirements for the resolution scheme. The URN requirement for persistence directly implies that the resolution service must cope with both mobility and evolution. Mobility in this case reflects the situations where users and resources may physically move over the lifetime of the URN, and indeed the services themselves may move. Evolution accepts that, over time, the infrastructure will evolve in terms of algorithms, protocols, file-systems, authentication etc. In general any resolution scheme must assume that any part of the infrastructure is subject to change and that any resource is subject to mobility.

The implication of the URN functional requirement of independence is that the resolution scheme must support delegation and isolation. Delegation allows the responsibility for assigning URNs and resolving names to be partitioned and passed on to other separate entities. Isolation is needed to permit entities, to which authority has been delegated, to operate within their delegated domain without reference to peers or their delegator.

Thus the major requirements for a URN resolution scheme are: mobility, evolution, delegation and isolation.

Figure 4.1: *Resolver Discover Service.*

URN resolution can be broken down into two steps; finding a resolver for a URN, and then resolving the URN. Finding a specific resolver for a URN requires a Resolver Discovery Service (RDS) as shown in figure 4.1. This thesis does not extend to a discussion of the proposed architectures of an RDS[4], which is an active and complementary research area [47, 95]. It is sufficient to assume at some point such a service will become available.

URNs solve the naming problem, and use of the RDS integrates the superset of URN schemes. However, the location problem remains unaddressed, and solutions to the location problem are domain specific. The remaining parts of this chapter are dedicated to documenting the design of the Nomad location service. From the point of the naming system, the Nomad location service fulfils the role of URN Resolver for the Nomad context.

4.2 Existing Location Services

Location service designs have previously organised themselves along geographical and organisational lines. This section reviews the Domain Name Service (DNS), which translates domain names into machine IP addresses, and is the most prominent example of a functioning scalable global location service. The other location service discussed is Globe, which is a contemporary research location service for a large scale network of distributed objects. The final part of this section details the approach of Flexinet, which is not intended as a global location service — but has a useful set of basic mechanisms.

[4]It is important to note that the RDS need not cope with the high degrees of mobility as found in Nomad, but rather cope with the lesser mobility of NID resolvers.

These examples serve two purposes, firstly they provide a means to introduce the problems that the Nomad location service architecture is designed to overcome, and secondly they provide a background against which the new architecture can be set.

4.2.1 The DNS

The DNS [71, 23] is a global naming service which returns IP addresses in exchange for a textual name.

The structure of the domain name-space is a hierarchical tree, based on a mixed geographic and organisational basis. For example, universities in New Zealand may register under the .ac.nz domain, where .ac represents the organisational, and .nz, the geographical element of the name. This structure is reflected in the design of the distributed database system which implements the DNS. It is this distributed database system that is of most interest in the context of this thesis.

The DNS database is distributed by dividing up the name space into non-overlapping zones forming a distributed search tree. Each zone has one authoritative[5] primary name server and a collection of secondary name servers, which hold a copy of all the records for that zone. The zone boundaries and the number of name servers are determined manually.

When a name server cannot locally resolve a query, it immediately contacts the appropriate top level domain name server (e.g. nz). The query is then recursively passed down through the tree until an authoritative record is located. The result propagates back to the local name server, where the record is cached for use by local queries until it expires.

The DNS is designed for slow changing machine:location bindings. In New Zealand, for example, the top level zone files are only updated twice a day. Caching and replica name servers reduce the query load on the root of the service, but the DNS is unable to update rapidly enough to manage short term object mobility.

Mobile IP [102] is a stopgap technology which sidesteps the DNS update delay by utilising redirection and tunnelling from the original location. There are a number of design issues that Mobile IP does not address, such as location independence or transparency, and in addition Mobile IP requires manual configuration and leaves residual dependencies on the system of origin.

[5] Records cached outside the zone are not authoritative.

4.2.2 Globe

The Globe [5, 106, 104] location service is also based on a hierarchical tree structured distributed database, however lookups are resolved from leaf to root rather than root to leaf as with the DNS, and are split on a purely geographic basis.

Each query starts at the local leaf node with the object ID, and recursively checks each node on the path to the root node. If an intermediate node has a record of the ID, then forwarding pointers are followed down the tree until the leaf node that contains the desired location information is reached. If the root node is reached and the ID is still unrecognised, then the object is not registered with the location service.

The flaw in this simple structure is that the root is the ultimate Object ID authority, and must therefore contain all Object IDs and associated forwarding pointers. In addition, the root needs to handle vast numbers of search and update accesses. This is addressed in Globe by partitioning the workload via a quad tree mapped over the surface of the Earth. Each partition implements a logical root to reduce communication lengths and areas of high load are partitioned more finely. Additional heuristics are applied in the nodes to cache geographically stable addresses. However, mobile objects are expected to migrate to be closer to clients, and to obtain execution and other resources. Mobile objects are therefore not expected to exhibit a high degree of geographic stability, and this is likely to reduce the gains from root partitioning and the caching heuristics. Additional heuristics in [3] attempt to address this problem by estimating the regional boundaries within which a mobile object remains.

4.2.3 Flexinet

Flexinet [40] is a java based object oriented middleware research architecture which supports amongst other things, cluster mobility. Unlike the DNS and Globe, Flexinet does not provide any form of global structure, geographical, organisational or otherwise.

A feature of Flexinet is that objects are represented by proxies and transparently rebound whenever the target object's cluster has moved. It is this rebinding of proxies which is utilised in the Nomad location service, the differences between the two approaches lie in how the binding is resolved.

In Flexinet, objects are created in a cluster on a 'Place', a restricted form of vHost, and each place provides a default static mobileNamer. Contrary to the implication of the name, a mobileNamer is not mobile, but rather it names (resolves) mobile clusters. When a cluster is created on a Place, it is assigned that Place's mobileNamer. From then on this mobileNamer is used as the resolver by proxies bound to objects in this cluster. This reliance on the Place of creation remains regardless of where the mobile cluster may move to, and constitutes a permanent and significant residual dependency and is a single point of failure. Removal of a specific Place will result in failure of all mobile clusters resolved through it.

Clusters may be created utilising a mobileNamer other than that provided by the Place, however these must not be mobile themselves – so all the problems with residual dependencies remain.

4.3 The Nomad Location Service

The DNS Globe and Flexinet all have limitations with high degrees of mobility. The shortcoming of the DNS in terms of handling mobility, Globe in terms of its reliance on geographic stability and the lack of global infrastructure in Flexinet reveals the necessity for a new Location Service for use in Nomad. An additional and complementary context can be provided alongside existing systems, utilising the URN RDS and fitting within and utilising existing and planned services.

Nomad contrasts at this point with other approaches to global or wide area location services. Globe's Wide-Area Location Service [106, 104] provides any user with access to any object on the Internet and ignores the alternative services that are or will be available. This is accomplished at the cost of excessive storage requirements for its top level nodes. Other middleware location services such as those offered by CORBA and Java RMI do not address the question of mobility.

A modern location service must track the set of locations for each mobile object throughout that object's lifetime. A seemingly simple task perhaps, but in a large scale distributed system a location service may have to deal with trillions of objects, some of which will be moving frequently. Scaling a system to this degree requires wide distribution, load balancing and a minimum of coordinating network traffic.

4.3.1 Exploiting Application Locality

The first two systems discussed in section 4.2 have one thing in common: a reliance on geography — either based partly on country codes as in the DNS, or totally as with the quad tree mapping in Globe. The major argument behind Globe's use of a geographical structure is to provide locality of reference. In highly mobile systems such as Nomad this no longer holds true, as application objects are envisioned as using their mobility to move closer to clients and resources. Instead there is another form of locality which can be exploited — the locality of reference within an application. If you interact with an application object once, chances are that you are quite likely to interact with it, or with another object from the same application, again.

The Nomad location service endeavours to take advantage of this and builds the basis of its location service around the concept of the application. That is, each application is wholly responsible for tracking and locating its objects for itself, the Nomad system and its clients.

There are two obvious and immediate objections to this simple and attractive idea. Firstly, the additional complexity imposed upon the application programmer, and secondly the need to rely on applications to correctly maintain their location information.

The issue of complexity is resolved in Nomad by the provision of a default Nomad location table class. When the application is created, a default location table object for the application is automatically and transparently instantiated. An application may also choose to override this transparency and substitute its own class for the default location table to satisfy its functional requirements[6]. In either case, once the application location table is created, it is updated automatically when any of the application's clusters move. The only direct control that the application need have is in determining the choice of host for, and the degree of replication of, the location table(s).

The transparent location updates ensure that the location table always receives the correct location information — the only way that an application can negatively impact the reliability of the system is by providing a faulty substitute class, or by choosing unreliable Depots and an insufficient degree of replication for the location table(s). These concerns are exactly the same as those experienced with regular application objects, and in any

[6] Consistency, failure detection and recovery, indexing and implementation details.

case, loss or corruption of the location table(s) is likely to be terminal for the application. Clients may suffer if they are interrupted during, or cannot obtain, a service. Clients may choose to respond by utilising a more reliable service in the future.

In addition to providing locality of reference, application location tables exhibit a high degree of inherent load distribution (they are as distributed as the applications themselves), and permit the application to tailor its location table to suit its needs and reference characteristics.

4.3.2 Design Overview

The Nomad location service must meet two different requirements; the discovery of services and the resolution of out-of-date bindings. Discovery can be considered as an external interface, while rebinding occurs internally and transparently. These two systems act independently but synergistically and are illustrated (without the RDS) in figure 4.2. In addition, this figure highlights the separation between the Nomad level services above the dividing line, and the application level ones below.

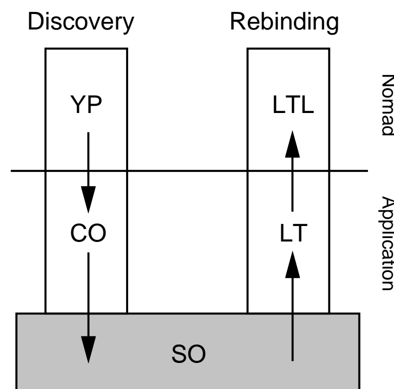


Figure 4.2: *Discovery and rebinding are complementary parts of the location service.*

The following list is a brief overview of the various location service components which feature in figure 4.2:

- **YellowPages (YP):** The YellowPages is the Nomad URN resolver, returned by the URN RDS from figure 4.1. The YellowPages takes a textual URN string and returns a reference to the contact object representing the service named. This is a Nomad level service.

- **Contact Object (CO):** Part of the application responsible for negotiating with the client to provide needed services. Once negotiation is successfully completed, the contact object returns a service object interface to the client.

- **Service Object (SO):** This is any object within the application which provides a contracted service to a client. It is to this object that the final binding is made.

- **Location Table Object (LT):** Contains references to all the mobile clusters composing an application and is responsible for tracing their movements. The location table has its own mobile cluster.

- **Location Table Locator (LTL):** The Nomad location table locator is a well known service responsible for locating and tracking the mobile location tables, and is a Nomad level service.

- **Local Interface Repository (LIR):** This is not part of the location service as such, rather it is part of the Depot architecture. It is maintained by the local Depot and acts as a source of useful references, such as the LTL. The LIR may be accessed by both applications and vHosts resident on a particular Depot.

As an example, consider a client which holds a textual URN for a service which it needs to use. This requires the discovery facet of the location service and the first step is to find a resolver for the URN using the RDS. Once the RDS system identifies the YellowPages as the Nomad URN resolver, the name is resolved returning a binding to an application specific contact object. The service is then provided to the client via a service object returned through the contact object. During negotiation with the contact object, or service via the service object — the target object may move, breaking the binding.

Broken bindings are resolved using the rebinding part of the location service. When a binding breaks, the subsequent invocation on that object will fail. The proxy representing the binding will transparently rebind via the object's cluster's location table. As the location table is also a mobile object, it may also have moved, requiring the rebinding to resort to a yet higher level — the location table locator (LTL).

A point worth emphasising is that each mobile cluster has a reference to a location table object, which is shared by all mobile clusters belonging to the same application. The location table object, which is in its own mobile cluster, has in turn a reference to the

global location table locator. All objects created with a mobile cluster share the same location table or location table locator.

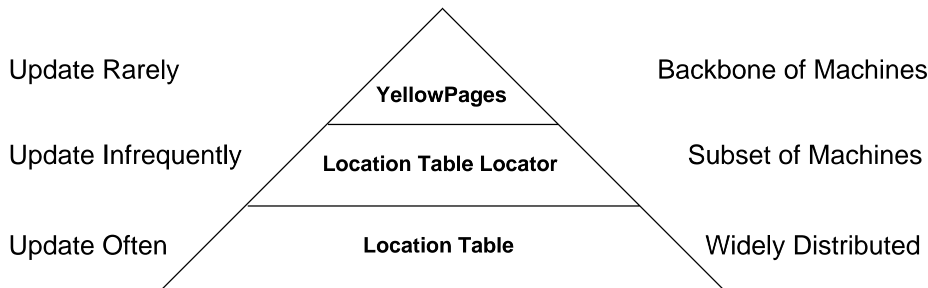


Figure 4.3: *Division of Labour.*

Figure 4.3 illustrates the division of labour between the components of the naming and location service resulting from this location service architecture. It is worth emphasising that as the load increases towards the bottom of the pyramid, so does the degree of distribution and replication.

The remainder of this chapter is dedicated to discussing the components that have been introduced in this chapter in greater detail. Sections 4.3.3 through 4.3.7 are arranged following the different phases of an application's life within the system, that is, its creation, registration, discovery, mobility and rebinding.

4.3.3 Creating an Application

Let's first consider starting an application from outside Nomad. To do this, a launcher negotiates for an initial vHost on which the application can be started.

The launcher then invokes the createApplication method on the vHost, triggering the events detailed in figure 4.4. The vHost starts by creating a cluster for holding the new application's location table, and sets the resolver[7] for the location table cluster to be the location table locator. The location table object is then instantiated inside the new cluster. Each location table is required to implement the `locationTable` interface, from appendix A.1.

The vHost next creates a cluster for the application object and sets its resolver to the newly

[7]Each mobile cluster has a resolver.

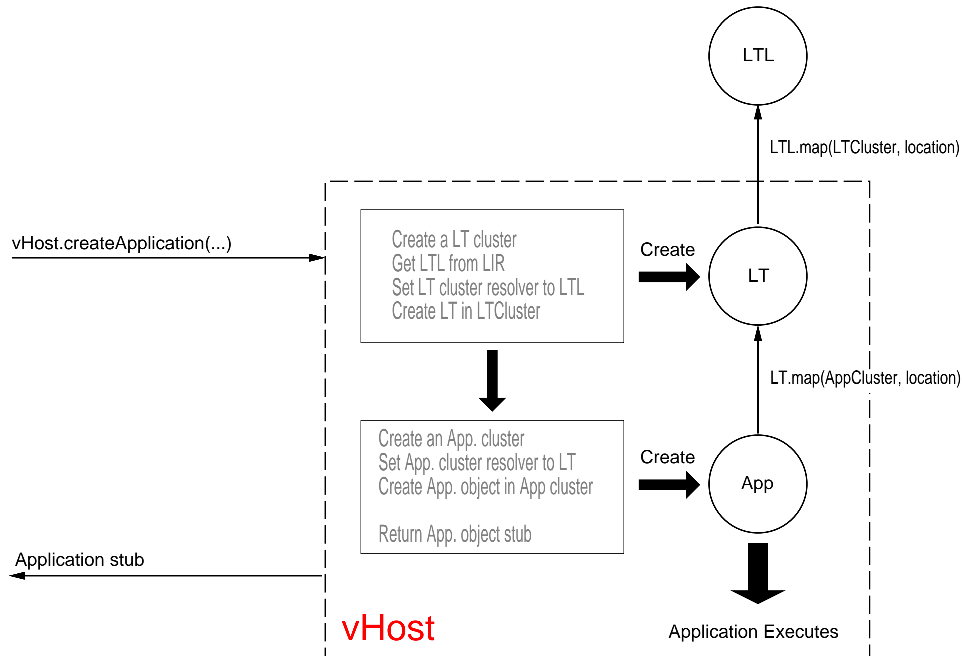


Figure 4.4: *Creating an application in Nomad*

created location table object. It then instantiates an object of the specified application class in the application cluster and returns to the launcher the interface (if any) on the application object. From this point the application will start to execute and construct itself independently of the launcher.

The steps for one Nomad application to create another application are similar.

4.3.4 Registering a Service

The creation of the application will not automatically create a corresponding entry in the YellowPages. It is now up to the application to register a URN, or it can simply remain anonymous to the outside world and respond only to the holder of the interface returned upon its creation. This private state of affairs may be desirable, for instance, when an application parent wishes to create an application purely for its own use, or to exert some control over the distribution of its interface. Otherwise, the application then registers itself through the steps illustrated in figure 4.5.

The application starts by obtaining a reference to a contact object representing itself, usually by creating it. The next step is to find the YellowPages with which the application needs to register, by using the Depot's LIR to acquire a reference to the YellowPages. With

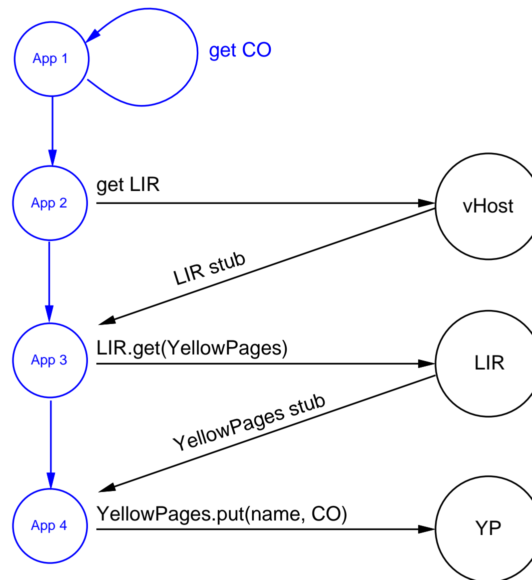


Figure 4.5: *Registering a service*

the YellowPages stub, it then registers the URN::CO association.

There is no dictate over what mappings are registered, combinations could include; one URN to one contact object, multiple URNs to one contact object, or multiple URNs to multiple contact objects.

As an example of what may be registered, consider the following URN which an application wishes to register: `urn:nomad:apps/currencyconverter`. The associated contact object may offer different levels of service e.g. gold, silver and bronze, which will then return the appropriate service object. Alternatively, the application may register three different URNs each with an associated contact object offering a different service level e.g. `urn:nomad:apps/currencyconverter/gold`. The contact object is still required to negotiate payment etc.

A significant advantage of the separate rebinding hierarchy now becomes evident — it does not matter if the contact object references held by the YellowPages are outdated, as they will be automatically rebound the first time they are invoked and found to be incorrect. This means that unless the application changes its contact object, the YellowPages does not need to be immediately updated to reflect changes due to mobility of the contact objects. Instead, the YellowPages can utilise a lazy, best effort update as omissions, delays and inconsistencies will be caught and then corrected by the rebinding system.

Section 4.1.1 detailed the URN functional requirements of independence, and section 4.1.3 detailed the corresponding resolution requirements of delegation and evolution. As a result the YellowPages can issue and resolve those names without recourse to peers or a delegator outside the YellowPages resolver itself.

Left unanswered in this thesis is the issue of entitlement, that is, whether a specific application is entitled to a particular name – this is subject to further complementary research involving URN naming issues.

4.3.5 Obtaining a Service

Figure 4.6 illustrates the four steps and seven messages which are needed to obtain a service from an application when starting from a textual service name. The example given here shows the initial binding to a Whiteboard application — from outside the Nomad system. From within the system, the RDS step would be unnecessary as the reference to the YellowPages is available from the LIR, the external access is shown here for completeness.

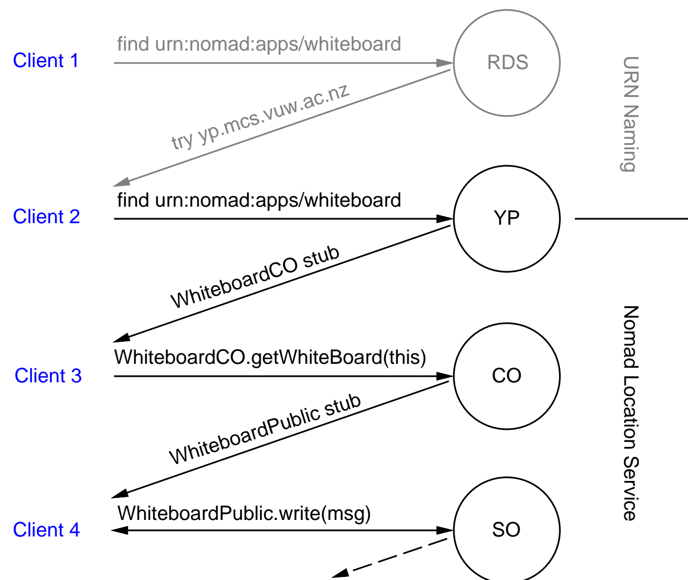


Figure 4.6: *Locating an Object.*

The first step involves the discovery of the URN resolver that resolves names for the domain `urn:nomad`. In Nomad the URN name resolver is the YellowPages, to which a reference is returned by the RDS. The client then queries the YellowPages with the same URN, and in reply gets a reference to the Whiteboard contact object. The methods which

a client can invoke on a contact object depend upon the application — in this case it is a request for a shared Whiteboard. The Contact object returns a WhiteboardPublic stub to the client which is an interface on the service object and is used to write to the Whiteboard. Normally there would be additional negotiation with the contact object — that is, requesting a schedule of charges and then invoking the method reflecting the chosen service level. This Whiteboard application is seen again as a fully worked example application in section 8.4.

Clearly all four steps need only be performed once during interaction with a service. However, when dealing with mobile objects it is possible that an object will have moved since it was last referenced. This then requires rebinding to occur transparently, as detailed in section 4.3.7.

4.3.6 Moving an Object

From the point of the location service, moving an object is very similar to creation. That is, the location table responsible for tracking the mobile object has the map method invoked on it. This simply updates the internal data to point to the new location. This call to update the location table is hidden within the mobility code, and no explicit action is required from the mobile object.

4.3.7 Rebinding an Object

As seen in section 4.3.6, updates to the location table and or location table locator are made automatically during migration. This does not however, solve the issue of all outstanding references held by others which no longer reflect the correct location. Figure 4.7 illustrates what happens when one such outdated reference is used to invoke a method on the remote object.

In figure 4.7, the initial invocation by the client results in the proxy attempting to contact the object on the vHost at which the cluster was hosted when the proxy was last used. The vHost communications stack attempts to invoke the remote method call on the object, but fails as the cluster in which the object resided no longer exists on this vHost. An error is returned to the invoking proxy – which then contacts the location table responsible for tracking this cluster (the reference to the location table is cached in the proxy). The

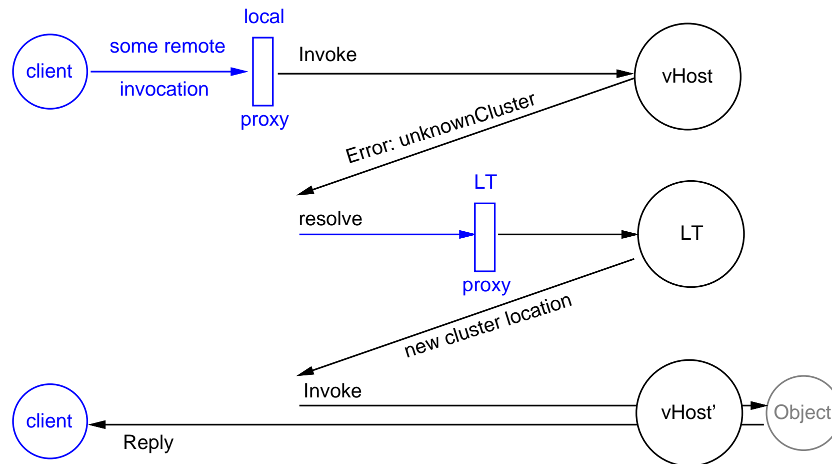


Figure 4.7: *Rebinding to an Object.*

location table replies with the new cluster location and the proxy reissues the original invocation to the object on its new vHost and caches the new location. Note, that as shown in figure 4.7, the call to the location table is also a remote invocation via a proxy. Since the location table is also mobile, this may also result in a cascaded rebinding, which is shown in figure 4.8.

In figure 4.8, the call to the location table by the location table proxy fails as the table has moved from vHost' since the last invocation via this proxy. As in figure 4.7, this results in an error being returned to the communicating proxy. The location table proxy then acts in exactly the same way as in the previous rebinding and uses its cached reference to contact the location table locator responsible for tracking the target location table cluster. The new cluster location is returned, the LT location updated within the proxy, and the original resolve reissued to the location table. The location table then returns the new cluster location to the original proxy, and the original invocation is reissued.

On both these occasions the rebinding occurs without action from and transparently to the client. Also, only the invoked reference is updated. An optimisation can be applied to ensure that all references to this cluster within a common vHost also reflect the change, however this may be problematic. For example, if the updated object then moves to another host — is this new value propagated to those present on the new host, or is the correct new value overwritten by the older value on the new host? At the very least, the update would need be timestamped — which raises the issue of clock synchronisation. While it may be worth caching updated references, they should only be treated as hints,

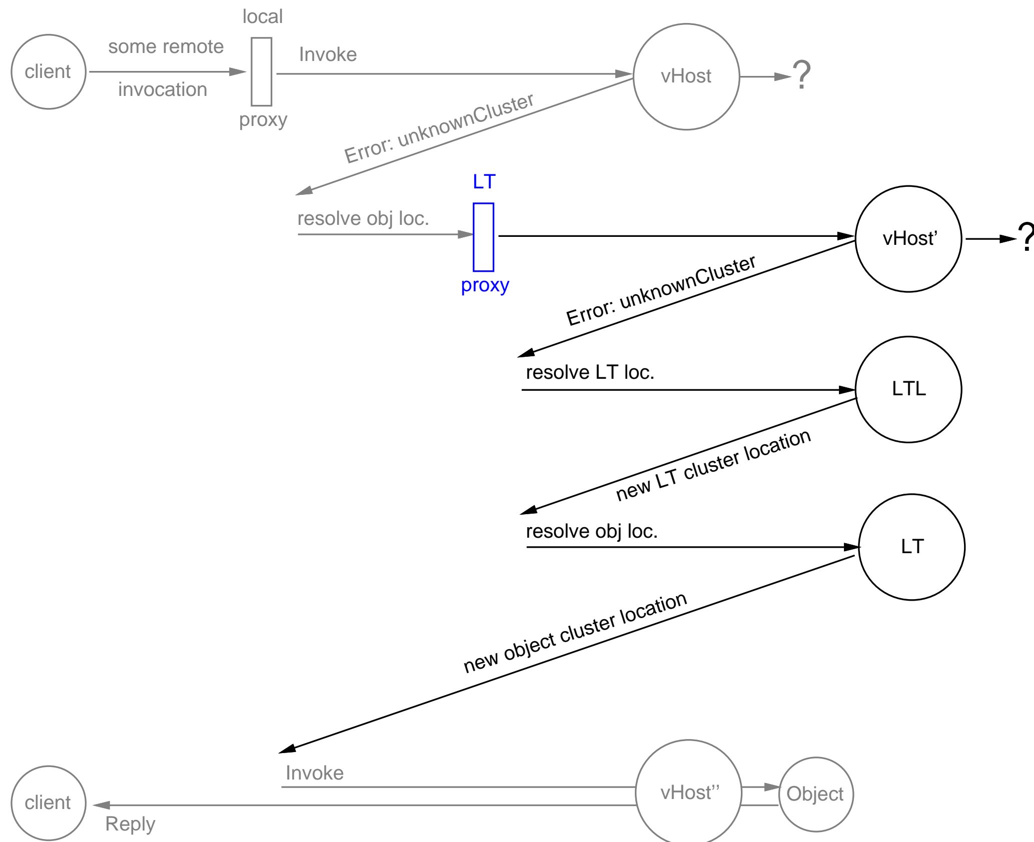


Figure 4.8: *Cascaded location table rebinding.*

and care taken to ensure that time that could be spent rebinding directly, not be wasted on exploring out-of-date hints.

It is worth summarising a few cases where the transparency can no longer be maintained:

- In the case where the **service object no longer exists**, the same steps as shown in figure 4.7 take place, except that the rebinding fails. At this point transparency can no longer be maintained and the failure is handed back to the client. How the application recovers a lost session is of course application dependent, but would need some form of negotiation performed by the contact object.
- Where there is a **complete failure of the application**, and the rebind through the location table locator fails to find a valid location table, then assuming the application can restart, all steps 1-4 from figure 4.6 need to be repeated. As with the situation where the service object no longer exists, the contact object negotiation will be needed if there is any hope of recovering a lost session.

4.3.8 Replication of the Location Tables

An application's location tables are owned by the root application object. Although the exact number of location table replicas is determined by the application policy, it is assumed that some degree of replication will be desirable. Relatively static location of the location tables is encouraged to reduce the burden on the location table locator.

Each cluster maintains references to more than one of the application's location tables. When a cluster is created it is initialised with a copy of the parent's location table references. When a cluster moves, the underlying migration system notifies the cluster's authoritative primary location table, which updates the replica location tables with a Bayou [25] like lazy consistency protocol.

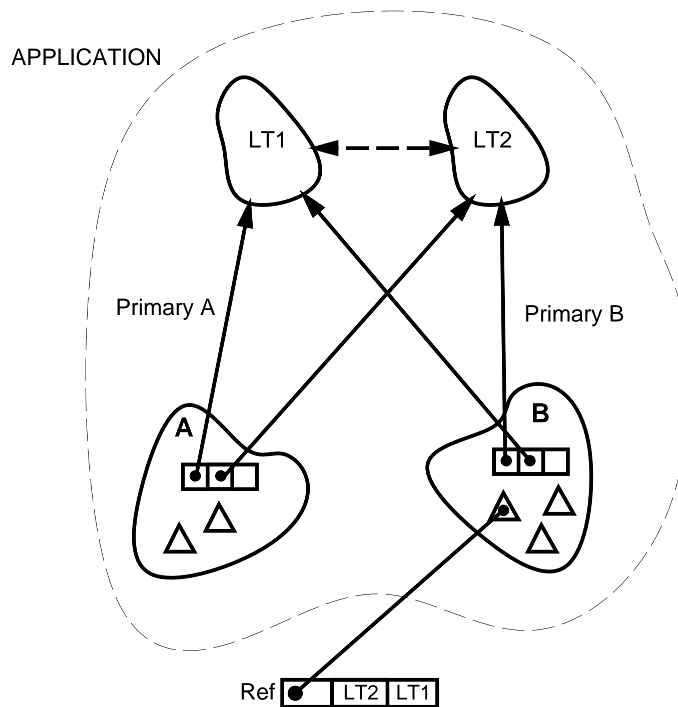


Figure 4.9: *Interplay of replicated Location Tables.*

Each cluster has one primary and a set of secondary location tables. When the cluster moves, the updates are made only to the primary. The primary is then responsible for updating the replicas, and only the primary location table of a cluster is authoritative. Each reference to an object in a mobile cluster contains the same ordered list of location tables, as shown in figure 4.9.

If a cluster cannot contact its primary location table, say during migration, the next location table in its list is promoted to be the primary, and this location table now propagates any changes. Subsequent resolves, by holders of references with outdated primary information, will update their existing reference to reflect the change of primary.

In the case of a network partition, two different location tables may consider that they are the primary — the original primary (P') and the newly promoted primary (P"). P" and the cluster must lie on one side of the network partition, and any references will be updated correctly as previously described. A reference on the other side of the partition will fail to contact the cluster and contact P', which only holds outdated information even though it considers that it is the primary. Since P' cannot supply any information that will allow the reference to breach the network partition, a fault condition results. Once the partition is repaired, a timestamp on the location table entry for each cluster ensures that P' is correctly updated and is demoted properly. If a primary location table cannot resolve a cluster at its last address, then it is assumed that the cluster no longer exists and is garbage collected from all the location table replicas. This is consistent with the situation involving network partition.

The promotion and demotion mechanism for primary location tables not only allows hot-swap location table replacement in fault conditions, but can also be used for load balancing over the set of replicated location tables.

4.3.9 Access Rights

Only the application has the authority to change the mappings within its location table, and this is done transparently during mobility. Likewise, the entry corresponding to the location table within the location table locator, may only be updated by the application – specifically in this case, by the application's location table. There are no access controls on resolving in any of the parts of the location service, and updating proxies to reflect a relocated object's cluster is undertaken by the proxies themselves on the client's behalf, not by the application.

4.3.10 Internal Architecture of Location Table Locator and YellowPages

The internal structure of the Location Table Locator and YellowPages are research problems in their own right, and are not within the scope of this particular thesis. However, the provision of location tables in the Nomad architecture reduces the load of mobility by the number of objects within an application. Therefore suitable candidates could include:

- The modified RDS B-tree architecture from [95] for the YellowPages.
- The location service architecture from Globe [5, 106, 104] could also be used for the location table locator with improved efficency — as only one reference is needed in the root for each application location table, rather than one reference for each object.

Implementation of garbage collection of dead entries in the LTL and YellowPages relies on the mechanisms designed into whichever architecture is adopted. However, the binding of service names to applications are likely to represent considerable investment on the part of the owner of an application. Such names therefore cannot be garbage collected and need to be rented (on say an annual basis) with right of renewal as is currently the situation with DNS entries. The LTL entries are a different matter being a purely internal representation, so they may be deleted when found to be invalid. Globe [3] itself takes this approach.

4.4 Fault Model

It is often impossible to tell if a failure is permanent or transient: a failed machine looks identical to a network partition involving that same machine. In all cases the Nomad location service assumes that the failure is permanent and purges disconnected entries from the LTL and the application's LTs. Entries in the YellowPages, which bind service names to applications, represent considerable investment to the owner and are not purged, as described above.

Section 4.3.8 outlines what happens when a location table is separated from the rest of the application (with or without fellow application clusters) by a network partition. Ultimately the location tables on each side of the partition become authoritative for any

clusters on their side of the partition, expunging all entries which are no longer reachable. Once the partition is repaired the application location tables can be merged consistently. This mechanism keeps the location service consistent, however, it is up to the application to deal with the failure and reintegration of its other components.

The correct application response depends on the components that have been disconnected, some can be freely abandoned (either not crash recovered or left to being garbage collected by their host Depot at the end of the current contract), whereas others represent longer term commitments, such as persisted clusters. One possible solution is that if a critical component finds itself marooned, it could create a local location table using the information already in its possession. This location table could reintegrate the component into the application on reconnection.

Crash failure is handled in the same way by the location tables, but without the need for reintegration.

Communication transparency between a disconnected, or crashed, object cannot be maintained in light of such failures. In each case, the communication failure is handed back to the invoking application, as described in section 4.3.7.

4.5 Summary

This chapter presents a novel solution to the design of a distributed location service for large scale mobile object systems. This approach uses the application to optimise the distribution of the location tables, and limits the impact of the majority of updates to these very small infrequently mobile data structures. The remaining global workload involves only resolving the locations of the location tables themselves.

The application architecture, separating the roles of the contact and service objects, along with the use of application specific location tables, encourage an application to intelligently distribute itself to meet negotiated client QoS requirements.

Chapter 5

Negotiation

In order to execute, applications need to acquire resources from Depots. The Nomad management architecture must therefore provide a negotiation mechanism that enables resource allocations between applications and Depots. This chapter will first focus on what negotiation is, and then what the architecture must provide to facilitate negotiation.

> **Negotiate**: *to hold communication or conference (with another) for the purpose of arranging some matter by mutual agreement; to discuss a matter with a view to some settlement or compromise.* - The Oxford English Dictionary.

From this definition we can extract the two essential components of negotiation; communication and compromise. For communication, the negotiating parties must share a common language and forum - this is the *mechanism* for supporting negotiation, i.e. how negotiation happens. The act of compromise indicates that participants are capable of identifying their common position to reach some form of agreement - essentially the *policies* behind the negotiation, i.e. what is negotiated and why.

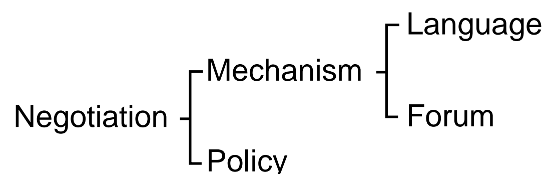


Figure 5.1: *The general composition of negotiation.*

Figure 5.1 illustrates this distinction between the mechanism needed for negotiation to occur, and the policies which define the compromise that is reached during negotiation. The

focus of this thesis is the mechanism, rather than the policies – as policies are dependent on the management goals of individual applications or Depots.

The mechanism can be further refined to make a distinction between the *language* which describes the content of a negotiation and the *forum* in which negotiation takes place. This chapter concentrates on the forum by discussing the relationships between entities in Nomad in section 5.1 and exploring the two general negotiation scenarios, bipartite and multipartite in section 5.4. The multipartite auction protocols are then further analysed in sections 5.4.2 through 5.4.7 revealing that within certain limitations the Vickrey auction protocol is ideal for automated resource negotiation.

The language is detailed in chapter 6, and chapter 7 presents the design of the Nomad Marketplace mechanism.

5.1 Relationships Between Participants

The next point of consideration is how the various entities in a negotiation interact. That is, what is the relationship between them. Sandholm [93] describes the range of negotiation relationships as cooperative, self-interested and hostile, which he describes only loosely via analogy. However, Sandholm's terminology is somewhat counterintuitive and inconsistent in the case of the self-interested relationship. A preferable definition is that a self-interested entity will be cooperative or hostile, depending on its expected payoff, and it is this definition that will be used in this thesis.

- **Cooperative (global gain, local loss):** Entities will attempt to increase the social welfare (see section 5.2) of the system - even if it results in a local loss.
- **Hostile (global loss, local neutral):** Entities will attempt to reduce the social welfare of the system - even if it results in a local loss.
- **Self-Interested (local gain):** Entities will attempt to maximise their local payoff, and in doing so, may be cooperative or hostile.

It is also worth stressing that these characterisations only apply during the process of negotiation - that is, the stance that the entities may take, and do not imply the trustworthiness or otherwise of the parties involved.

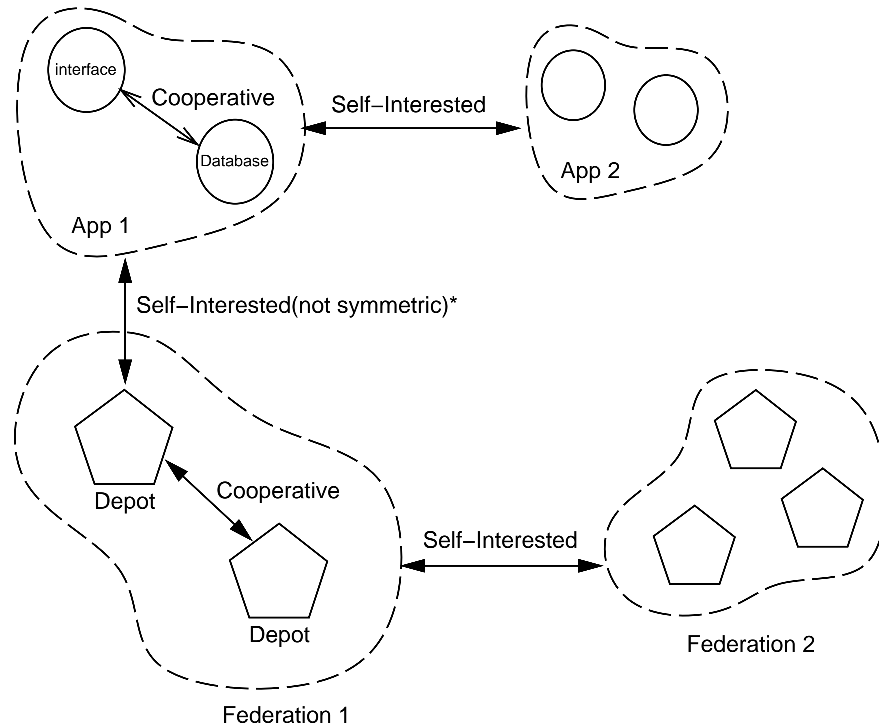


Figure 5.2: *Relationships in Nomad*

Figure 5.2 identifies these relationships between the various entities which exist in Nomad. Recall that each of the self-interested relationships may be either cooperative or hostile.

Previous systems such as Contract Net [97] were restricted to systems with cooperative elements. Contract Net was later extended to produce TRACONET [90, 91] which coped with selfish[1] bidders. If situations such as the *tragedy of the commons* [38] are to be avoided, then entities must be encouraged to behave in a desirable way. Obviously a Self-Interested entity can act cooperatively if it is in its own interest to do so, and that is what the Nomad architecture must achieve.

5.1.1 Potential Impacts of Hostility

It is reasonable to assume the worst in any open system, where applications may come and go without recourse to a central authority. Since the Nomad system is open and cannot restrict the range of permissible behaviours to non-hostile, the architecture must

[1]Sandholm called this Self-Interested, however this actually means that entities will not perform tasks for others without compensation and will reduce the utility of peers if they can benefit locally. They will however, stop short of reducing the utility of the whole system.

therefore be designed as if each of the Self-Interested relationships were in fact hostile. The potentially hostile relationships are:

1. **Application to Application**

   Hostility in this case can be considered to be competition. Applications may undercut each other, or even make a short term loss if the long term result is that a competitor is driven out of business. This is acceptable as long as applications cannot damage each other without incurring the real financial cost of their actions.

2. **Application to Depot**

   An application may misrepresent its resource needs to gain a more favourable price or allocation, and cause the depot to over-reserve or over-commit its resources. The application could also generate spurious negotiations wasting Depot capacity.

3. **Depot to Application**

   In the Depot to Application relationship we have a non-symmetric relationship (hence Self-Interested* in figure 5.2), as the Depot has considerable capability to damage an application. These are discussed under point 4.

4. **Federation to Federation**

   The Depots in a federation could discount and over-allocate their capacity to try and drive competitors out of business. This could mean that applications with valid contracts for a Depot would either be turned away - a form of Depot initiated denial of service, or they would receive fewer resources than were contracted for. Federations may also attempt to damage each other, for example, by sanction. A federation may refuse migrations from outside, or prevent applications from leaving once resident.

   These and associated problems require a mechanism to police contract enforcement. This is expanded on in section 9.2.3 and requires additional research.

5.2 Metrics of Negotiation

The ideal result of negotiation is mutual satisfaction. Game theory and economics [87, 93, 72, 26] have a number of measures of this, most notably; Pareto-optimal, social welfare,

stability, individual rationality and symmetry:

- **Pareto-optimal:** A distribution of resources is Pareto-optimal if any redistribution of resources which is beneficial to one or more individuals is also detrimental to one or more others.

- **Social Welfare:** The social welfare of a system is the sum of all individual payoffs in a given distribution of resources. When the social welfare of a system is maximised the distribution of resources is Pareto-optimal.

- **Stability:** A situation is stable if it is in every individual's interest to retain its current strategy. Therefore a negotiation protocol is stable if there is a dominant strategy [2]. Otherwise the Nash equilibrium [87] can be used to compute a measure of the stability of the protocol. However, some situations have no strategies that form a Nash equilibrium, while others have multiple Nash equilibria.

- **Individual Rationality:** Participation in a negotiation is individually rational if the payoff to the individual is not less than the payoff of not participating in the negotiation. In non-cooperative systems, all successful negotiations must be individually rational.

- **Symmetry:** A symmetric system is one where individuals with the same behaviour will have the same expected payoffs. That is, the system treats all individuals without bias.

5.3 General Negotiation Mechanisms

This chapter has so far focused on competition and compromise in negotiation. This has however resulted in the neglect of the most basic form of negotiation - the informational type, which illustrates that negotiation need not always be competitive.

[2]For the definition of Dominant-Strategies, see section 5.4.5.

5.3.1 Informational Negotiation

Informational negotiation is where participants simply want to agree on a basis for communication. It does not involve any form of compromise[3] and there is no concept of gain for either party - except the mutually beneficial improvement in communication.

Informational negotiation typically involves low level query-response exchanges between parties to establish a basis for further communication. The majority of negotiations are simple minded, as in selecting a protocol version or packet size. A good example of this is the option negotiation protocol for the TELNET Network Virtual Terminal (NVT) as specified in RFC854. This protocol allows customisation of the NVT by enabling or disabling NVT options to better suit the capabilities of the two communicating parties. If an option is not understood, then any request to enable it is automatically rejected. Unfortunately the only way of determining which options the other NVT implements is to try each one - a slow and expensive method. A better solution is the feature discovery protocol for FTP as specified in RFC2389. The basis of this mechanism is provided through the FEAT command. This modified FTP protocol permits a client to discover which additional commands are supported by specific servers. The mechanism is simple - upon receiving FEAT, the server returns a list of all commands and mechanisms not specified in either RFC959 or RFC2389.

The main benefit of the feature discovery mechanism is to reduce unnecessary traffic between client and server - without the FEAT command a client would have to try each new extension resulting in a flurry of network traffic and rely on correct server behaviour concerning unknown commands.

5.4 Competitive Negotiation

In Nomad the goal of negotiation is fair and dynamic resource allocation. This must be done within a difficult environment, where components are self-interested, untrustworthy, and negotiations are frequent[4] and concern low value[5] items.

There are two general styles of negotiation, those involving two interested parties - bi-

[3]Therefore requiring no intelligence.

[4]Negotiations are frequent as applications are highly mobile.

[5]Low value items require correspondingly lower overheads.

partite, and those involving more than two - multipartite. In the case of multipartite negotiations there are either multiple sellers and one buyer or one seller and multiple buyers. These two situations differ fundamentally, as in bipartite negotiations there is competition between the buyer and the seller, whereas in multipartite negotiations all the competition is between the buyers (assuming the latter case). This is illustrated in figure 5.3.

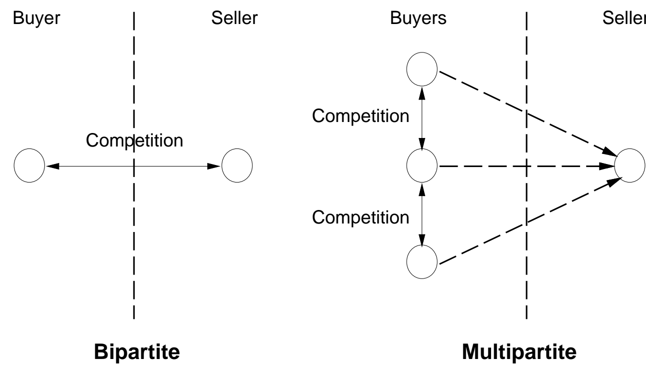


Figure 5.3: *Multiple buyers and one vendor introduce competition on one side.*

The different plane in which the competition lies affects the ways in which the information about an item under negotiation is revealed. In bipartite negotiation, both participants keep such information as private as possible - to encourage equal compromise on both sides. Whereas in multipartite negotiation, the seller specifies the item as completely as possible. In this case, competition between bidders is relied upon to ensure that the seller, in revealing its complete information, is not disadvantaged. These two approaches are discussed in considerable detail in sections 5.4.1 and 5.4.2 respectively.

5.4.1 Bipartite Negotiation

Bipartite negotiation is essentially a constraint satisfaction problem between one buyer and one seller. Figure 5.4 is an idealised graph from [57].

Constraints $C^a(x)$ and $C^b(x)$ define the individual areas of interest of parties a and b, $C(x)$ defines the area of common interest (the intersection) and Po is the Pareto-optimal solution. As x increases solutions are less satisfactory for a and more satisfactory for b. In non-cooperative environments both a and b must proceed from points of ignorance to determine the intersection, otherwise:

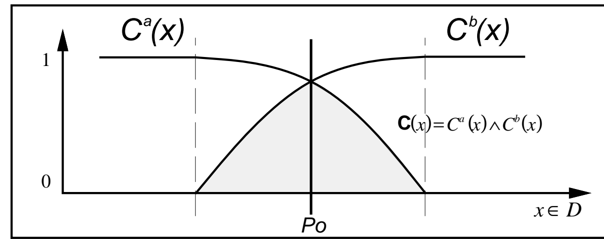


Figure 5.4: *The area of common interest.*

1. the constraints containing sensitive information should not be released to the public domain,
2. knowledge of another's constraint space enables cheating, and
3. lying may be encouraged, leading to sub-optimal allocations.

An independent third party to oversee or audit the completed negotiation, as suggested in [54], is just as easily misled as a fellow negotiator.

However, without full disclosure, bipartite negotiation suffers from a number of shortcomings. There is a high message overhead while the negotiators attempt to discover the areas of mutual agreement, so it is expensive for low value items. Where the item being negotiated is composed of the multiple components, the valuations the negotiators place on the separate components may result in non-optimal allocations. Indeed what was intended by one party as a concession may look like a hardening of position to the other.

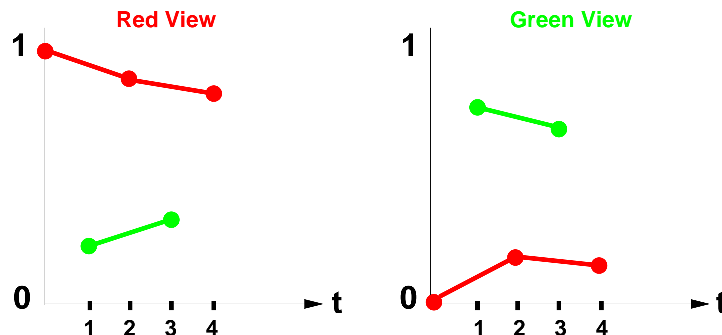


Figure 5.5: *Non-homogeneous component valuations can result in non-optimal allocations.*

Figure 5.5 illustrates this situation with the left graph being the red negotiator's viewpoint, and the right representing green's view of the negotiation. In the above scenario, the red negotiator has made a concession at t = 4 in response to green's offer at t = 3. However,

as red and green value the components of the item being negotiated differently - to green, red's offer at t = 4 looks like a hardening of position. This is a local minimum and without some form of disclosure or algorithmic solution (such as the costly probabilistic 'bounce' introduced to gradient descent learning in neural networks [58]), will result in a sub-optimal solution. A non-optimal solution will result in a negative impact globally as well as to the individual parties.

The last and possibly most important shortcoming for our environment involves heterogeneous systems. In this case the buyer cannot determine if the seller can satisfy its constraints prior to completing the negotiation, e.g. you cannot buy apples in a tyre shop. The obvious solution to this is advertising, however this amounts to full disclosure if fruitless negotiations are to be avoided.

This analysis demonstrates that bipartite negotiations are susceptible to cheating and sub-optimal allocation, which makes them unsuitable for automation. Bipartite negotiations are appropriate for high value, low volume items - such as secondhand cars. The most appropriate scenario would see the bipartite negotiator used as an expert system requiring that a human supervise the commit phase.

5.4.2 Multipartite Negotiation and Auction Protocols

Multipartite negotiation takes place via auction which is an efficient distributed mechanism for solving task and resource allocation problems. There are four main types of auction protocol identified by William Vickrey [107]; the English, Dutch, Sealed-Bid, and what has since become known as the Vickrey auction protocol.

1. **English (open ascending)** The bids start low and progressively increase by open outcry until only one bidder remains. The winner then claims the item at the price of their last bid.

2. **Dutch (open descending)** The auction starts the item at a high value and the auctioneer continuously lowers the price until one of the bidders bids. The bidder wins and takes the item at the current price.

3. **Sealed-Bid (sealed-bid, first price)** Each bidder submits a single sealed-bid. The bids are opened simultaneously and the winner is the bidder with the highest bid.

4. **Vickrey (sealed-bid, second price)** The bidders each submit a single sealed-bid. The bids are opened simultaneously and the winner is the bidder with the highest bid, who claims the item at the price of the second highest bid.

Analysis in [107] reveals that the English auction protocol is more certainly Pareto-optimal than the Dutch auction protocol, especially in the case where bidders are heterogeneous or unsophisticated. The sealed-bid auction protocol is equivalent to the Dutch auction protocol, and suffers from the same shortcomings with heterogeneous or unsophisticated bidders. The Vickrey type auction protocol is a Pareto-optimal version of the sealed-bid auction protocol, and is equivalent to the English auction protocol subject to the eight caveats detailed in sections 5.4.7.

5.4.3 Valuation

An auction consists of an auction protocol, auctioneer, client and a set of potential bidders. The analysis of an auction protocol depends partly on how bidders' value the items being auctioned.

- **Private value:** In private value auctions the valuation depends solely on the bidders own preferences. That is, no account is taken of potential resale value. This valuation scheme does not normally model human auctions well. A example would be auctioning a lease on a rental property - where the lease is explicitly non-transferable and sub-letting is forbidden.

- **Common value:** The valuation in a common value auction depends entirely on the other bidders values of the item. In this case, no bidder prefers the goods, the value comes entirely from resale possibilities. An example of this form of valuation is the wholesale apple auction. In this case, bidders are only interested in what they will be able to resell the apples for, they have no interest in consuming the apples themselves. Of course, this example assumes that all lots are homogeneous.

- **Correlated value:** Correlated valuations are a combination of private and common values. In these cases the bidder may choose to keep or on-sell the goods. This is the model that is normally selected for modelling human auctions.

5.4.4 Auction Equivalence

The most significant result in auction theory is the **revenue equivalence theorem** [107]. This states that all four auction protocols yield the same *expected* return in private value auctions.

In addition, the strategies and payoffs associated with the Dutch and Sealed-Bid auction protocols, even for non private-value auctions, are the same. That is, the Dutch and Sealed-Bid auction protocols are strategically equivalent, in all three valuation models, as only the winning bid matters and no information is revealed during the auction process.

In private value auctions the English and Vickrey auction protocols produce the same allocations (at the same prices), where the bidder who values the item most wins it, but do so with different strategies, see section 5.4.5. However, the English and Vickrey auction protocols are not equivalent in non-private value auctions, as the open outcry nature of the English auction protocol provides additional information to the bidders, which can then alter their valuations. Milgrom and Weber [67] show that in correlated value auctions, this enables the English[6] auction protocol to generate greater revenue than the Vickrey, Dutch and Sealed-Bid protocols when there are three or more bidders.

5.4.5 Dominant Strategies

If a certain strategy pays a player the highest playoff, regardless of other players' strategies, then that strategy is known as a dominant strategy.

Neither the Dutch nor the Sealed-Bid auction protocols have a dominant strategy. The dominant strategy for the English auction protocol is to bid a small increment over the current bid price and stop when the private value is reached. The dominant strategy for the Vickrey auction protocol is to bid the true value of the item. Recall, the Vickrey auction protocol is won by the highest bidder, but at the price determined by that of the second highest bidder. Bidding less than true value can only decrease the chance of winning and cannot have any effect on the price paid, as this is set by the second highest bidder. Bidding more than the true value to increase the probability of winning could involve the bidder in an unprofitable transaction. For example, say A has a true value of

[6] When the protocol is **open exit**, that is, bidders are aware when each bidder ceases to bid.

10 and B has a true value of 9. If B bids 11 in order to win the auction, then it pays $A's$ bid price of 10, which is one more than $B's$ true value of 9 and hence incurs a loss.

5.4.6 The Benefit of Truthful Bidding

Truthful bidding is the dominant strategy for the Vickrey auction protocol, which has two important and beneficial side effects:

- bidders reveal their values accurately allowing for globally efficient allocations, and
- bidders need not waste efforts in attempting to counter-speculate other bidders.

Counter-speculation in itself would waste computational resources and introduce considerable complexity into the system, while not improving the overall allocation.

In addition, there is the obvious efficiency of submitting a single bid, based on the stable description of lot[7] and auction. That is, not only does the bidder submit just a single bid - but they do not need to be kept up-to-date about the current state of the auction in a timely and reliable fashion, as they would with the English or Dutch auction protocols. This minimises the number of messages exchanged during an auction.

The Vickrey auction protocol is a one shot Pareto-optimal technique that is highly efficient in terms of messages and allocation. This suggests it is ideal for machine to machine negotiation and is the auction protocol selected for use in Nomad.

It is worth pointing out that even small modifications to the protocol may seriously damage its dominant strategy of truthful bidding. For example, in Miller and Drexler [31, 69] and Spawn [109], the addition of the *escalation* mechanism reintroduces counterspeculation and non-truthful bidding. The basis of the escalation modification is to prevent starvation, by linearly increasing all bids over time, so that later bids need to be higher to compete with earlier bids – therefore, the longer a bidder waits, the higher its priority. Counterspeculation occurs as a bidder now needs to concern itself when it and its competitors bid. The strategy is not to bid the true value as early as possible, as earlier lesser bids will still win. The result is that bidders bid less than their true value and bid early, if they are bidding late, then they should bid their true value. Therefore there is no

[7] A *lot* is an individual set of goods in an auction, which are bid on as a unit.

dominant strategy in the modified version, and it should not be used in non-cooperative systems.

5.4.7 Protocol Limitations

There are however, a number of limitations concerning the applicability of the Vickrey auction protocol. These have arisen in the literature [87][92][107], and need to be addressed in the design of the Nomad Negotiation Mechanism, which is presented in chapter 7.

Bidder Collusion

All four auction protocols are susceptible to bidder collusion, however the English and Vickrey auction protocols self-police such pacts. The following example from [87] demonstrates this.

Assume bidder A has a true value of 20, while bidders $b_1 \dots b_n$ have true values of eighteen or less. Bidders $b_1 \dots b_n$ agree to bid 5, while A bids 6. In the English auction protocol, due to its open outcry nature, A can observe if any of bidders $b_1 \dots b_n$ breaks the pact, and can retaliate by increasing its bid. Since $A's$ true value exceeds that of any of the bidders $b_1 \dots b_n$, and therefore A can outbid them, the defaulter gains nothing and is best to remain in the pact.

In the Vickrey auction protocol, A bids its true value of 20, $b_1 \dots b_n$ bid 5, and on winning A pays the second bid price of five. Since bidders $b_1 \dots b_n$ can gain nothing from bidding higher than the pact value, there is no incentive to break the agreement. This of course requires that bidders are able to identify one another for the purposes of arranging the collusion.

The Lying Auctioneer

The lying auctioneer problem stems from the winner paying the price of the second highest bid. If the auctioneer's profit is based on a percentage of the sale price, then it is potentially to its advantage to lie about the value of the second placed bid and therefore gain a higher commission. The other three auction protocols are immune from this problem, as the winner pays the amount that it bid.

Signing of the bids by the bidders, as suggested in [92], can prevent the auctioneer from fabricating the second price bid – if the auctioneer is required to submit the second price bid to the winner to authenticate the auction. However, this solution has two main problems as it:

- reveals who another bidder was, possibly enabling future collusion, and
- reveals the second bidder's *private* true valuation to a competitor.

Sensitive Information Possibly Released

As with the constraints in bipartite negotiation, true values may represent sensitive information. Bidders may consider it important that their true values are not released into the public domain. A compromised auctioneer or bid transport system could result in the release of this information.

Lying in non-Private Value Auctions

In non-private value auctions the value is dependent on the the price that the other bidders place on the item. This means that if a bidder wins the auction, then it has paid too much. This conundrum is called the *winner's curse* and applies to all auction protocols, as it is an issue concerning the basis of valuation rather than the protocol.

The net result however, is to alter the dominant strategies. In the case of the Vickrey protocol, the new dominant strategy becomes one of bidding less than true value in order to avoid a loss.

However this situation only applies where the item is to be resold into the same market. An example of this would be a wholesaler who can then on-sell at retail.

Lower Revenue in non-Private Value Auctions

As mentioned in section 5.4.4 on auction equivalence, in correlated or common value auctions, the open outcry nature of the English auction protocol allows bidders to increase their valuations during the auction. In situations with three or more bidders this leads to a higher revenue for the open exit[8] version of the English auction and therefore a

[8] Bidders signal that they are exiting the auction, and may not reenter.

comparatively lower revenue in the other three auction protocols.

Sub-optimal Allocation and Lying in Interrelated Auctions

If an auction involves heterogeneous interrelated lots - then independent Vickrey auctions of the lots may result in globally suboptimal allocations. That is, it may be optimal for one bidder to win all the interrelated lots, but that bidder is prevented from doing so by a single winning bid by another bidder.

Solutions to this problem can involve lookahead, which includes counterspeculation, and consequent lying to obtain the globally optimal solution. This is essentially subsidising the bid on one lot with the expected gains from the other interrelated lots.

Uncertainty of Valuation

If a bidder cannot compute its true valuation - due to complexity, lack of computation resources, or time constraints, then the dominant strategy changes if the bidder is risk averse. That is, the risk averse bidder is better to bid less than its estimate of its true value rather than its true value. This is obviously no longer a truthful bid.

Wasteful Counter-speculation

Where there is an uncertainty of valuation, and a bidder b_1 can pay to obtain an exact valuation v_1 for price c, then [92] finds that b_1 should buy the valuation iff $v_2 \geq \sqrt{2c}$, where v_2 is the competing b_2 valuation. In this case b_1 gains from counterspeculating b_2.

However the above analysis requires a number of critical assumptions; that the b_2 valuation of v_2 is certain, that v_2 is common knowledge for b_1 and finally that c is itself not uncertain.

5.5 Summary

The Vickrey auction is an ideal basis for automation - provided that its limitations are addressed in the design of the protocol mechanism. The distribution is Pareto-optimal [107][87][67] and consequently also maximises social welfare. The protocol satisfies the requirement for stability, by having a dominant strategy, while the system is individually

rational, as truthful bidding ensures that the payoff is not less than not participating. That is, even if the second bid is equal to the winner, the winner will at worst incur no loss. Lastly the system is symmetric, as the same behaviours will result in the same payoffs.

Chapter 7 details the design of the Nomad Marketplace, a mechanism based on the Vickrey auction protocol, which overcomes the limitations detailed in section 5.4.7.

Chapter 6

Representation of Resources

Chapter 5 identified the process and language aspects of negotiation, and detailed these processes of negotiation. This chapter explores the means of representing what is being negotiated, that is, the language. The representation of resources is critical to the versatility and optimality of any auction negotiation protocol, and without it, there cannot be a reasonable auction mechanism.

6.1 Representations

The majority of standard auction mechanisms are concerned with the sale of a single good and therefore represent each item separately. This is fine when the good is a single unit, such as a soccer ball, but not when the good in question is, say, one sock — which has its greatest value when part of a pair. This synergy may not always hold, for example, individual socks may have greater value to a sock puppet factory than as part of a pair of socks. Likewise, execution resources form an indivisible set, related and conditional upon the availability of each other. Piecewise negotiation of these individual resources will not give any usable result let alone optimal allocations. After all, it is very difficult to execute when the memory is on a different host from the allocated CPU cycles.

6.1.1 Utility Function

One way of combining separate components into one unit is the utility function. This is a mathematical function which ranks alternatives according to their utility to an individ-

ual. This takes several individual components and combines them into one term using coefficients to indicate significance.

$$\alpha A + \beta B + \gamma C$$

This form makes it easy to compare different utilities, as each is represented as a single numerical value, but lacks flexibility and cannot cope with boolean values. The utility function may also introduce ambiguity in negotiations, where the likely use of different coefficients by bidders can result in an increase in apparent value for one bidder, but a corresponding decrease in the valuation for another. This can result in sub-optimal local minima as shown in section 5.4.1, when full valuation information is not available. In addition, the utility function implicitly assumes that no specific resource is essential, that is, it cannot express dependencies, i.e, "I only want B if I also get A" [76]. For example, a bidder may have no C, yet still have the highest utility if the coefficients are weighted towards A and B. These problems are part of a bigger problem concerning combinatorial allocations.

6.1.2 The Combinatorial Allocation Problem

Combinatorial allocation (CA) [89, 77] reflects the situation where a set of components have a synergistic value that exceeds the sum of the individual parts. Because of complementarities and substitution effects, bidders have preferences not just for particular items, but for collections of items. The combinatorial allocation problem (CAP) is to allocate items to maximise total value over all bidders. Let there be $i = 1, \dots, I$ bidders, and each bidder selects some bundle of goods S_i for which it has valuation $v_i(s_i)$. The set of all selections is $S = (S_1, \dots, S_I)$, and this set of all selections without i is denoted by S_{-i}. The solution to the CAP is S^*, the maximising set from S based on the valuations of the bidders:

$$S^* = max \sum_{i=1}^{I} v_i(S_i) \quad : \quad \forall_{i,j} i \neq j \; and \; S_i \cap S_j = 0$$

There are two facets to the solution of the CAP, the first is *bid-expression*, that is, how to communicate bid combinations and valuations from a bidder to the auctioneer. The second

problem is *winner-determination*, that is, the allocation problem. Winner-determination is *NP*-complete, and is equivalent to the maximum weighted set packing problem [77]. Considerable research, particularly in the fields of management and operations research has been done in winner-determination, and combinatorial auctions have been held for FCC spectrum rights, delivery routes, and matched airport time slots [28].

Generalised Vickrey Auction

The generalised Vickrey auction [65] (GVA) is an extension of the original Vickrey protocol to implement allocations of multiple goods, multiple units of goods and externalities[1] for quasilinear[2] utility functions.

The basic idea behind the GVA is that each winning bidder pays their bid less a discount. This discount is a computation of the impact of a bidders participation in the auction, that is, the increase in overall value that the bidder has directly contributed.

The GVA first solves the CAP finding S^*, the maximising allocation over S, and the maximising valuation V^* of S^* computed from the bidder utility functions $\bar{v}$. Set S^* has W winners where $W \subseteq I$. The GVA then solves the CAP an additional W times removing each winning bidder $i \in W$ in turn to find $(S_{-i})^*$ and corresponding valuation $(V_{-i})^*$. The difference in valuations, $V^* - (V_{-i})^*$ measures the contribution to the valuation V^* by bidder i and is called the Vickrey discount (Δ). The subsequent amount that a winning bidder i pays for their allocation S_i is their valuation less Δ_i:

$$\Delta_i = V^* - (V_{-i})^*$$

The winner's final price is their bid $v_i(S_i^*)$ less the discount:

$$price(i) = v_i(S_i^*) - \Delta_i$$

This is exactly equivalent to the original Vickrey auction for a single good. The discount Δ_i is the contribution to the valuation V^* by i and once this is subtracted it reflects the

[1] Externalities, are factors outside the direct process of the auction that influence bidders' valuations. For example, if one agent cares that another wins the auction.

[2] Quasilinear functions are of the form: $v(x_1, x_2) = g(x_1) + x_2$.

second-price valuation. However, the generalised Vickrey auction has one financial and two significant computational problems:

1. **Problem 1** The GVA is intractable for the auctioneer. It must solve multiple CAP instances, once for V and then V_{-b} for every bidder participating in the optimal allocation set.

2. **Problem 2** The GVA is intractable for bidders with hard valuation problems. All bidders must provide valuations for all possible combinations of items. Incomplete or approximate valuations may result in non-optimal allocations.

3. **Problem 3** There is no guarantee that the incomes and pay-outs (revenue) of the market will balance, in general they will not — this is addressed heuristically in [78].

iBundle

One recent heuristic approximation to the computational problems in the GVA is iBundle [76]. The iBundle protocol is an ascending price auction specifically designed for e-commerce that permits bidders to make OR and XOR bids over collections of bundles. Each bundle is a logically indivisible good during the auction.

The iBundle protocol is iterative rather than single bid, as in the GVA protocol. Each round involves the auctioneer informing the bidders of the current bid price. Bidders respond with a bid of the form: (b_1, p_1) OR (b_2, p_2) indicating that the bidder wants one or both bundles b_1, b_2 at prices p_1, p_2. The auction continues until the bids are unchanged from the last round, or all bidders are determined to be *happy*. A happy bidder is one that has received at least one bundle on which it has an XOR bid.

iBundle heuristically addresses the problem of incomplete bidder valuations, but does not address the intractability of the CAP, in this case the CAP must be computed once each round. Yet iBundle achieves excellent results and is in general capable of 99% of optimal, compared to the GVA at 100%, with much the much lower bidder valuation revelation of 71% (100 required for GVA). However, the basis on which bundles are formed is unclear in [76], as are the rules concerning the combination of the logical operators, e.g. combining $(A\ OR\ B)\ XOR\ (A\ XOR\ B)$ to provide an equivalent for $(A\ AND\ B)$. Thus the allocation problem of "I only want B if I also get A" remains unanswered.

6.1.3 Prećis

There are a number of problems with existing auction protocols for handling multiple allocations and constraints. While existing research marginally considers bidder constraints, no allowance is made for vendor constraints. The reasonable approach is to consider the CAP as a mutual constraint satisfaction issue, this focus is absent from previous research. For example, an auction involving military surplus might well have the vendor constraints that: only foodstuffs be sold to neutral countries, munitions only be sold to allied countries, and absolutely no sales to hostile countries. A bidder constraint on the same auction could be: "I want Tanks, but only if I also get spare parts".

In addition, the GVA has clear problems with computation — having to complete an *NP*-complete computation once for the allocation, and then again for each bidder in the allocation. The GVA auction must also be centralised for optimal allocations, and bidders must compute valuations for all possible combinations of allocations. Other limitations with the GVA are detailed in Wellman [111].

The iBundle auction protocol avoids some of the intractability problems of the GVA auction heuristically, but in order to do so resorts to an ascending bid auction. The computation required of the iBundle auctioneer is still high, in that it must compute the provisional CAP in each round, and must evaluate *happiness* and retain all bids and allocations between rounds. Additional effort is required to compute the next round of prices, possibly including discriminatory prices.

6.2 Resource Description Graph

The approach in this thesis is to change the emphasis from that taken in prior research. Rather than directly addressing the *NP*-complete winner-determination problem, the focus is instead on minimising it, through encoding of resource constraints within the bid-expression.

In Nomad vendor constraints do not reflect existing resources for sale, but rather resources that are required by the vendor and supplied by the bidders — a form of tender. In addition it would be preferable that the bidders compute their valuations directly, rather than requiring the auctioneer to compute them — certainly the bidders will be more

distributed and therefore have, collectively, more computing power available than the auctioneer.

These requirements are met by the resource description graph (RDG) developed for describing and valuing resources in Nomad. The essential idea is one of valuing paths through a graph describing the resource constraints, as shown in figure 6.1.

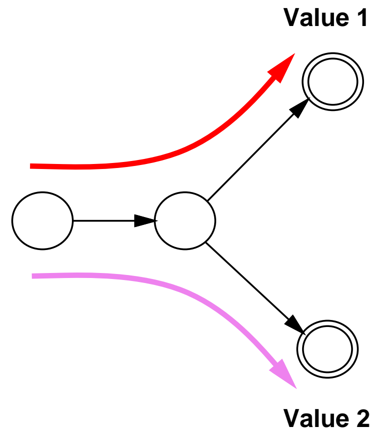


Figure 6.1: *A valuation is applied to each path (sentence) through the RDG.*

6.2.1 The Graph

A RDG is a rooted directed acyclic graph consisting of vertices and edges. Edges represent individual resources and vertices indicate if they are an accept state. Each accept state represents a task which the vendor wishes to perform, such as replicating a service object or performing a difficult computation. The resources defined by the edges in the path, from the root of the RDG to the accept state, form the *sentence* of resources which are required to perform that task. Each accept state may have one or more sentences, that is, an accept state does not uniquely identify a sentence.

The RDG from figure 6.2 has two accept states, and five resource requirements. There are three distinct sentences: $e1, e2, e3$ is the sole sentence for state 3, and state 4 has the two sentences $e1, e2, e4$ and $e1, e5$. Each sentence has a valuation, and the valuations of two sentences to the same accept state are unlikely to be equal as they represent different resources. Any combination of resources not defined by a sentence to an accept state is invalid. This eases the winner-determination problem by tagging acceptable and invalid resource combinations. In addition the RDG not only describes the common area between

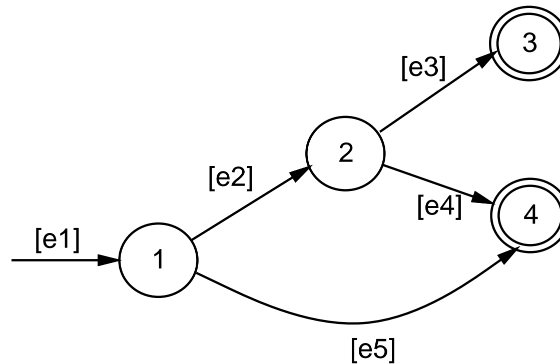


Figure 6.2: *A simple RDG, double circles indicate accept states*

entities, it also describes compromise with multiple sentences to accept states.

Bids are only made on accept states and the following graph structures in figures 6.3, 6.4 and 6.5, map to logical combinations of AND, OR and XOR:

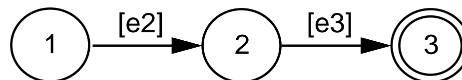


Figure 6.3: *($e2$ AND $e3$)*

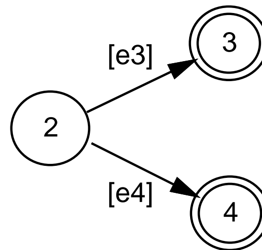


Figure 6.4: *($e3$ OR $e4$)*

In figure 6.3, bids on accept state {③} must satisfy both (and) $\{e2,\ e3\}$. Figure 6.4 has two accept states {③, ④} and a bidder may bid on {③} or {④} or both, for which the bidder must respectively satisfy $\{e3\}$ or $\{e4\}$ or both $\{e3,\ e4\}$. Lastly, figure 6.5 has one accept state {④} for which two alternative sentences exist, i.e., only one task, but it can be satisfied by either of two different sets of resources. In this case, the bidder may bid once on {④} with either both $\{e2,\ e4\}$ or $\{e5\}$.

These basic logical equivalences of the RDG demonstrate that the RDG has *at least* the

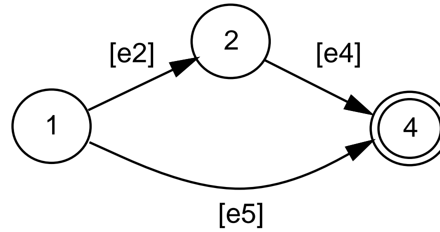


Figure 6.5: *((e2 AND e4) XOR e5)*

expressive power of iBundle [76]. Recall, expressions represent requirements and not actual resources as with the sale of goods. Therefore, if a bidder bids on two accept states with a common non-existential[3] resource in their sentences, then that bidder is implicitly declaring that it has multiple units of that resource available. Internal vertices in the RDG may also be accept states as multiples of resources can exist to satisfy the intersection of sentences.

As the RDG is an acyclic graph, there are no loops and therefore this possible route for attack by malicious applications can be discounted. The RDG is a more expressive combinator than the bundles from [76], as the combinations are formed based on vendor requirements and not bidder preferences. In this way the RDG reflects the synergy of the components for the vendor, and properly reflects its resource requirements.

6.2.2 The RDG Combinatorial Allocation Problem

The application RDG captures the relevant synergy from the application's point of view. This is however, not the same as synergy from the bidders' point of view. For example, assume each accept state corresponds to one application object. These objects share a common set of classes which will require loading. Clearly the cost of hosting one of these objects could be mitigated by hosting additional objects from this same application as the classes only need to be loaded once for multiple instances. The solution is to also use an RDG to express the bidder's own bid constraints (BBC). As an example, consider that the application RDG is as shown in figure 6.2, and that a bidder only wants accept state ③ if it can also have ④. The complementary BBC graph is shown in figure 6.6.

[3]For example, Java 1.1 is a an existential resource, whereas 100Mb of memory is not.

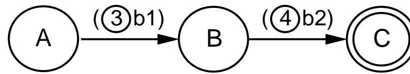


Figure 6.6: *The bidder's bid constraints.*

Here the bidder encodes in the BBC its preference that it only wants its bid b_1 on ③ made if it also obtains ④ for b_2. The inclusion of a BBC allows the auctioneer to dabble in discriminatory pricing, for example, if the bid b_1 is low, but the overall BBC bid $b_1 + b_2$ is higher than the sum of the other bids, then accepting the BBC bid in effect gains ③ for less than the price apparent to other bidders.

Once the auction is over, the auctioneer uses both the RDG and BBC graphs to compute the final allocations. This iteration is only performed once, as opposed to iBundle, and the winner's price for each accept state is computed in the the same way as in the GVA, that is, once to solve the CAP, and then again for each bidder in the allocation set. The number of combinations satisfying the RDG and BBC will usually be less than, or at worst equal to, the number of combinations solved by the original GVA.

This is a very general mechanism, but one that still requires the auctioneer to compute an *NP*-complete allocation multiple times. For Nomad, a simplification is needed, as the computational cost of the optimal auction cannot be justified by the intrinsic low value of the resources being auctioned. That is, a trade-off between optimality and computability is demanded. The solution is to simply *discard the BBC*, and allow each bidder to *make an entirety bid* over all of the lots in an RDG. Entirety bidding, described for human auctions by Milgrom [45], is a coarser means of capturing synergy, but is sufficient to address examples such as the auction of a bankrupt manufacturer; where the value of the land, plant and equipment may, for some buyers, exceed the sum of their individual values.

The entirety bid captures a degree of synergy, but relies on the application (vendor) to create auctions with accept states such that they are attractive targets for entirety bids. It is clearly in the vendor's best interest to encourage this in order to gain a better price. However, the vendor can never know exactly what each bidder would gain synergy from, and this is the only loss. In contrast the gains from this simplification are significant. The combination of RDG and entirety bidding can be used with the original Vickrey single bid auction, giving a non *NP*-complete allocation, and approximate solution to the CAP. This approximation requires experimental validation, but this is left to future work and as

usual such analysis will reflect that there is a tradeoff between accuracy and computational efficiency. The real question is how optimal an allocation need be — and in the case of Nomad, with low value goods, the answer is probably not very. True practical experience with more than a prototype will be needed to answer this.

6.2.3 Graph Composition

Sophisticated applications could be programmed to construct each RDG from scratch, based on explicit or implicit policies and runtime environment. However, a significant advantage of the RDG is that it can be created from programmer defined building blocks that an application can merge as required. That is, each mobile cluster within an application may be associated with its own RDG. An application can then compose these individual RDGs into larger graphs to automatically reflect the clustering of the application's clusters on a single Depot. Likewise, these larger graphs can be later decomposed to reflect the dispersion of the clusters. The application may also respond dynamically to the runtime environment by adjusting values in the constraint expressions.

6.2.4 Versatility of Description Graphs

A significant advantage of RDGs is that they facilitate delegation of negotiation within an application in a natural and efficient way. The discussion so far has implied a 'central' authority within the application that understands the different roles of its components and contains a complete application RDG specification.

The Nomad architecture also enables decisions to be taken on a local level via delegated authority. For example, if a service object or other object becomes overloaded. One solution is, for it itself, to create a replica on a suitable Depot to share the workload. The advantage of the RDG is that as each component of an application is spawned, it can take its part of the application's RDG specification with it. Later replication occurring at this level can then be negotiated by the object itself, without referral to a central application negotiation coordinator.

6.2.5 Textual RDG Representation

The RDG is used for disseminating application resource information throughout Nomad. An RDG is created by a programmer to reflect policy, passed to the Market and distributed within catalogues to Depots. For these reasons, the RDG has a lightweight human readable textual form which can be stored, printed and distributed safely. It contains no code or active components via which attacks may be made upon the auctioneer or Depots.

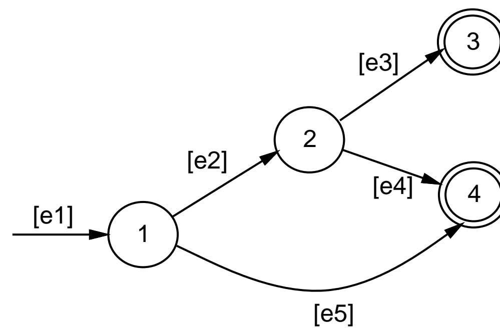


Figure 6.7: *A simple RDG example.*

Each node has at least one transition into it, a list of children, a name, and a boolean value indicating if the node is an accept state. For example, in figure 6.7 the root is reached by the transition containing the expression `[e1]`, is named `1`, is not an accept state (`false`) and has children `(...)` as shown below:

```
([e1] (1 false ((...) ())))
```

The textual RDG grammar is:

```
accept -> boolean
boolean -> "true" | "false"
complexExpr -> ~']'
expr -> '[' complexExpr ']' | '[' boolean ']'
name -> [alpha | digit]+
nodelist -> RDT nodelist | "()"
reference -> '#'name

RDT -> '(' expr RDTnode ')'
RDTnode -> '(' name accept '(' nodelist ')'')' | '(' reference ')'
```

The textual representation of figure 6.7 is:

```
([e1](1 false(([e2](2 false(([e3](3 true(()))) ([e4](4 true(())))
                              ())))
          ([e5](#4)) ())))
```

Multiple paths into nodes are encoded as a single transition (the first), and subsequent transitions as symbolic links to the unique node name. This is evident in expression $[e5]$, which has a symbolic link to node ④ rather than a normal edge.

The simplest expression is a boolean value, if that boolean value is false then the edge is pruned. Pruning continues back up the graph until an accept state or other sibling is reached. This provides a simple and automatic mechanism for an application to easily control what parts of the RDG it wants. For example, an application may have a general RDG which has subgraphs for various application components descending from the root. Simply by toggling the transition between true and false the subgraph can be included or excluded from the auction.

6.2.6 Edge expressions

The primary aim of the research into the RDG was to support the negotiation protocol mechanism. As a result of this focus, little has been explored in the direction of specific edge expressions, which represent policy (including QoS constraints). The provision of a useful set of resource expressions is a research project in its own right — including analysis of application requirements, programmer experience, and some deeper appreciation of likely Depot resource management policies. Any RDG edge expressions must conform to a standard Nomad convention, and the specification of edge expressions should not be 'open'. Otherwise the opportunity to introduce NP-complete problems to the RDG computation exist.

However, when constructing the simple proof-of-concept test applications in chapter 8 there was an opportunity to experiment with expressions for specifying location requirements. The two such expressions are:

- **near:** near($host$, $metric$) by some metric, say $ping < 50ms$. The host could be the location of a client, fellow application object, or some resource.

- **not:** not($host$), that is, exclude the host from bidding. This could be used to prevent co-location of replicas.

Another two location candidates which were considered were **at** and **between**. However the expression, at($host$), eliminates competition in the auction, enabling the specified host to realise it has a monopoly — this problem and the solution are discussed in greater detail in section 7.4. The expression, between($host_1$, $host_2$, $metric$), states that the bidder must satisfy $metric$ for both $host_1$ and $host_2$. This is equivalent to two consecutive near expressions and is therefore redundant. Direct experimental experience with these expressions was, as indicated in chapter 8, only with the near expression.

One caveat is that a RDG needs to provide sufficient information about an application's requirements before Depots can form valuations or reserve resources. The best means of encoding this significant amount of information is to use 'standardised' characteristics, such as heavyIO and a level qualifier — say 10. This is essentially a nickname for a common set of resource requirements. The remainder of the RDG is used to specify any specific divergence from the standard characterisation, and any boolean requirements, such as persistence, communications protocols, libraries etc. Figure 6.8 shows one such RDG, with the divergence from the standard characterisation being the *near* requirement.

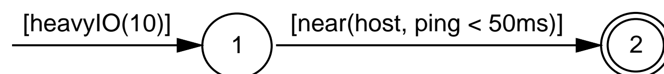


Figure 6.8: *An RDG using standardised RDG edge expressions.*

An advantage of well characterised standardised levels, in addition to compactness, is that if an application finds it is insufficient, it can simply increment or decrement its level qualifier until it finds the lowest level which is satisfactory. This eases the application's task of determining its requirements and although it is an approximation, runtime variation prevents exact specification in any case. The workload incurred by the Depots when bidding is also reduced, as there is less graph traversal, and well known characterisations can have corresponding predefined responses. For example, a Depot may wish to bid with a discount on all RDGs requiring highIO, and these are now easily identified.

6.3 Pattern Matching

Another advantage of RDGs is that they can be compared structurally. There has been much work in the area of computer vision, constructing trees or graphs representing an object, and then utilising homomorphisms and cliques to recognise specific images [8, 64, 80, 79, 81, 85]. These techniques can be applied to find matches between RDGs without costly transformations. The MarketPlace, in chapter 7, uses RDG matching to select target Depots for advertising. Of course this is yet another *NP*-Complete problem, but the set of algorithms developed for computer vision purposes are reasonably efficient approximations. As targeting of advertising need not be highly discriminating, quite crude approximations acting as market-side filters will suffice.

6.4 Summary

A resource description graph (RDG) is a single graph which describes an auction and associated compromises. Each sentence through the graph identifies a non-unique set of resources required to fulfil that task, alternate sentences define acceptable compromises. Multiple lots can be specified in an RDG and a bid over all the lots in an auction, an entirety bid, can capture a degree of bidder synergy.

The RDG provides a general representation of resources, which permits the vendor or bidder to limit the permissible allocation combinations, and reduce the combinatorial allocation problem (CAP). Nomad goes further and utilises the RDG and entirety bidding to replace the *NP*-complete CAP with a good approximation enabling multi-component auctions with the standard Vickrey auction protocol. This avoids the difficulty of balancing the books and computing the social welfare of the system, as the winner simply pays the second price. Privacy is enhanced as bidders need only reveal a single bid value to the Market (auctioneer). In addition, the Market need not parse and understand RDG edge expressions — this evaluation is performed in the Depots.

The RDG forms a natural basis for pattern matching, and can fully and compactly describe the resource needs of an application by encoding its divergence from a set of standard characterisations. The RDG is at least as descriptive as the iBundle logical combinators, but with a single bid auction rather than an incremental ascending auction.

Chapter 7

A Negotiation Mechanism for Nomad

If negotiation is too complex or time consuming, negotiators will be larger and will incur greater storage and computation costs, discouraging mobility. Applications running within Nomad exist to perform some service or task — their primary goal is almost certainly not one of sophisticated and costly negotiation. The resources being bid for are of low value, and available from multiple Depots. Therefore, it is not expected that any one auction is sufficiently critical to require human intervention. The best solution is a mechanism which enables applications with simple, computationally bound negotiation routines (simple intelligence) to survive, while not preventing more sophisticated behaviour.

Auctions have been discussed in chapter 5 with the familiar context where the vendor possesses the item and wishes to realise the highest price for it. For the resource allocation problem in Nomad, we turn this upside down. That is, the *vendor* (**the application**) wishes to obtain the item/service it has specified from a *bidder* (**a Depot**) — at the lowest price. This can be thought of as a vendor putting a contract up for tender, on which bidders make bids. Thus the auction is for the right to host an application and supply the set of specified resources. All the prior auction analysis still holds, it is just the mental image which must be reversed.

Chapter 5 has shown that the Vickrey auction protocol has the desirable properties of: Pareto-optimality, maximising social welfare, stability, individual rationality and symmetry. In addition, the Vickrey auction is also economic in terms of auction state and message

overhead. This chapter describes the mechanism design of the Nomad Marketplace, which addresses each of the limitations of the Vickrey auction protocol that were detailed in section 5.4.7, resulting in an enviably efficient basis for negotiation.

7.1 The Market

An independent auctioneer is needed to deal with the problems of a lying or compromised auctioneer. In essence:

- A winning Depot needs to be sure it has received the second highest price, but other bids and the bidders must be kept secret, and
- an application needs to know that Depots have not been in collusion resulting in an unfair bid price.

The only way to address this problem is to build an auctioneer which cannot be compromised. Thus we identify the need for a trusted third party application operating at the Nomad system level, the Marketplace. The Marketplace is comprised of local Markets — the structure of which is documented in the following sections. The global structure in which the local markets form the Nomad Marketplace is detailed in section 7.6.

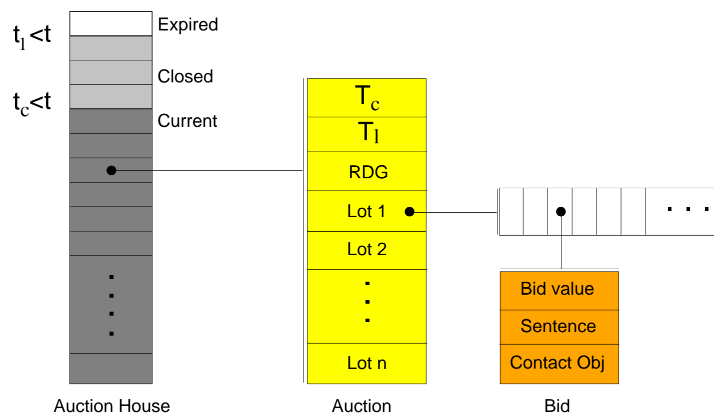


Figure 7.1: *The Market Structure.*

Figure 7.1 illustrates the significant data structures of a Nomad Market. The auction house is an ordered list of auctions, sorted on the auction closing time T_c. Each auction is placed by an application and contains one or more lots, a resource description graph (see

chapter 6) describing all the lots in an auction, and the time at which this auction closes T_c and the time by which any outstanding contracts lapse. Each lot has an ordered list of Depot bids on that lot, sorted on bid price and arrival time (to eliminate ties). Each bid contains a sentence (again, see chapter 6) through the resource description graph and a Depot manager contact object at which, if this is the winning bid, the application can redeem the contract. The second timer T_l is the time by which the aforementioned contract must be redeemed. T_l is essential if the Depots are to schedule resources efficiently and recycle expired reservations. Expiry of T_l also allows the market to free the auction slot by writing it to a log file.

From the point of a Depot, the shorter length of time that its resources are reserved for, the more bids it can participate in. It therefore makes sense for bidders to submit their bids at the last possible moment, before the auction closes at T_c. This is termed just-in-time (JIT) bidding. The Market encourages bidding Depots to indulge in JIT bidding by time ordering auctions on their T_c. After T_c the market accepts no further bids for an auction. JIT bidding has the following advantages:

- There is a shorter interval between bidding and a contract being redeemed — potentially giving the bidding Depot a better utilisation of resources.

- Urgent contracts are easily identifiable by having short T_c times, and as they appear they are bid on immediately.

- From the auctioneer's point of view, JIT bidding increases the number of bids, increasing competition as the bids are not fragmented over the entire auction space.

- The implementation of the market place is simplified as garbage collection of auction slots is automatic, and the closing and maintenance of auctions does not require extensive searching and updating.

An entire auction may be withdrawn by the application until T_c, and likewise, individual bids may be withdrawn by the bidding Depot until time T_c. After T_c all bids are binding.

7.2 The Negotiation

The market is itself a Nomad application, and as such differentiates between contact and service objects. When the local market is retrieved from the local interface repository — a contact object is returned, from which different service interfaces are obtained depending on whether the client is an application or Depot. The contact and service interfaces are included in Appendix A.2.2.

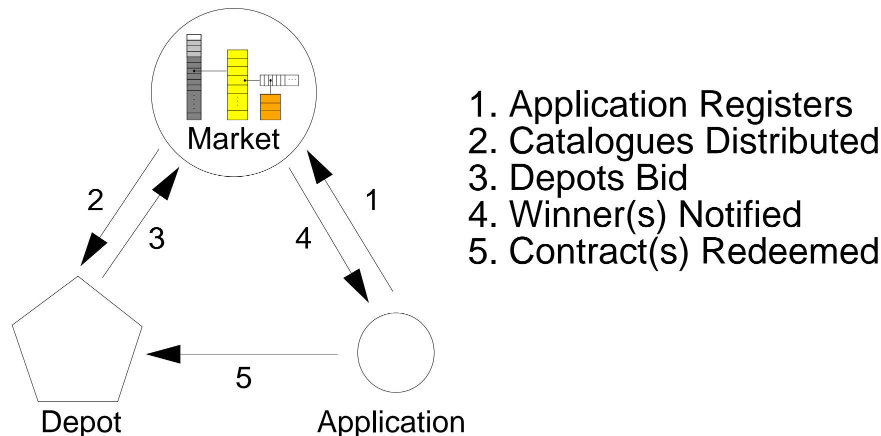


Figure 7.2: *Negotiating for Resources.*

There are five steps to complete a resource negotiation and these are illustrated in figure 7.2. Each of these steps is detailed in the corresponding 7.2.X sections 7.2.1 through 7.2.5. These sections assume that the Market has a pool of registered Depots. Depots register with their local Market, using the Market contact object, and may optionally include a resource description of the types of lots that they are interested in bidding on. Applications also register with the Market, however as they are expected to be mobile between different Depots using different Markets, no application state is kept between auctions.

7.2.1 Registering an Auction

An application places an auction with its local market with a message of the form:

`<rdg><vending interface><aProps>→aRef` where:

- `<rdg>` is the resource description graph describing the application's resource requirements. If a resource description graph has n accept states — then the auction

will have n lots.

- <`vending interface`> is the interface through which the application will be advised of the result of the auction.
- <`aProps`> is the set of properties which describe how the auction is to be conducted. There are three properties pertinent to this section:
  - <`type`> There are currently three types, standard, entirety (section 7.3) and bipartite (section 7.4).
  - $< T_c >$ The closing time of the auction.
  - $< T_l >$ The lapse time, by which any contracts negotiated in this auction will be redeemed.

  In addition there are other auction properties, detailed in section 7.7, which are used in the global version (Marketplace) of the market.
- The return value `aRef`, is a reference to the auction, which the application can later use to query the state of the auction, or prior to T_c, to withdraw it.

This information is then placed in the auction structure, shown in figure 7.1, and a catalogue created.

7.2.2 Catalogue Distribution

Catalogues are the advertising mechanism in the Market, and are distributed and customised in three ways. Firstly, when an auction is placed, the Market generates partial matches against all of the resource description graphs previously registered by the Depots; pattern matching is discussed in section 6.3. As targeting of advertising need not be highly discriminating, a crude approximation acting as market-side filters will suffice. Depots with matches are then sent a catalogue for this auction. Secondly, a Depot may request a JIT catalogue containing all auctions falling within a window of length Δ. Thirdly, a catalogue will be created and issued to a specific Depot for a simulated bipartite auction, as described in section 7.4. Catalogues are of the form:

[<`rdg`><`market`><`aRef`><`type`>$< T_c >< T_l >$]*; **where:**

- `<market>` is the Market at which this auction is being held. This is required as catalogues can be shared in the global Marketplace and the actual Market must be identified, see section 7.6, and

- the remaining message elements are defined as before.

The catalogue is sorted on T_c, and the `<type>` is sanitised to remove the bipartite type, see section 7.4.

7.2.3 Bidding

On receipt of a catalogue, the Depot decides if it wishes to make a bid. If so, the Depot evaluates the resource description, traversing its branches, pruning and finally determining its valuation on each reachable accept state. The resultant message contains all of the bids for each of the lots that the Depot is bidding on:

`<aRef><depot>` [`<entirety value><sentences>`^{n}]$^?$[`<bid value><sentence>`]* $\rightarrow$**bRef**
where:

- `<depot>` is the contact interface on the Depot from where, if the bid is successful, the negotiated service will be obtained.

- `<entirety value>` states if an auction is of entirety type, see section 7.3 for details. The entirety bid is optional, and the Depot may also submit bids on the individual lots.

- `<sentence>` is a unique sentence of resources which identifies the accept state and associated lot.

- `<bid value>` is the Depot valuation of a RDG sentence to the accept state and corresponding lot.

- `<sentences>`^{n} are the n sentences for each of the n accept states in the resource description graph, all of which must be included in a valid entirety bid.

- The return value `bRef`, is a reference to the bid, which the Depot can later use to query the state of the auction or bid, and prior to T_c to withdraw the bid.

- The remaining message element `aRef` is as defined previously.

If a Depot issues a new bid for a lot on which it has previously bid, then the older bid is automatically replaced with the new bid. An optimisation from [97] allows for an auction to close immediately once an acceptable bid has been received. However, this optimisation is susceptible to cheating in a competitive environment, where a Depot could progressively increase its bid until it is accepted. For this reason, auctions do not terminate early in Nomad.

7.2.4 Closing the Auction

When an auction closes, the Market computes the winner of each lot and winner of the entirety auction (if any). The Market next constructs a contract token message for the application which contains a signed contract token corresponding to each lot in the auction plus one for the entirety auction (if any). Each token contains the state of each lot, the contact object of the winning depot, the second price bid and the path(s) to each lot. The state indicates whether the auction was successful, that is, if there were at least two valid bids. If there was only one bid on the lot, the value returned is that of the winning bid as no second price bid exists. The application may still choose to accept it if it considers it reasonable, as negotiation is only binding on the bidders.

[<aRef><state><Depot><entirety 2nd value><sentences>$^{n} < T_l > \#SIG]^{?}$

[<aRef><state><Depot><bid 2nd value><sentence>$< T_l > \#SIG]^{*}$

Where:

- <state> can be one of three values; valid (2 or more bids), oneBid, and noBids.
- <Depot> is the contact object for the winning Depot.
- <entirety 2nd value> is the price from the 2nd place entirety bid.
- <bid 2nd value> is the price from the 2nd place lot bid.
- $\#SIG$ is the MD5 hash of the token, signed with the private key of the Market.

In the current design of the market, only the application receives the auction result notification. At time T_l any unclaimed reservations (unsuccessful bids) can be freed by the Depots. Design alternatives include also notifying the winner, or notifying all parties

participating in an auction. The justification behind only notifying the application is to minimise the number of messages. Further analysis is needed to determine if the cost of reserving Depot resources outweighs the gain from fewer messages. In a non-prototype system, all participants should be notified if the lapse time is significant.

7.2.5 Redeeming the Contract

Each of the contract tokens in the contract token message is separately signed. The application then redeems the individual contract tokens at their Depots and obtains a contract for the resources described in the associated sentence. The contract token is binding on the Depot, however the application may choose not to redeem some or any of the contract tokens. If this were not so, entirety bidding would not be possible — and extracting payment from a transient application is problematic. Imagine the following scenario; an application places an auction, and then awaits the result. The machine that it is executing on fails, and prevents the application from redeeming its contract tokens. Who should pay, the application or the faulty host? It is not clear that there is a reasonable answer let alone an algorithm for automatically determining culpability. The best solution is to make the contracts non-binding on the application. This could open the Market and Depots to abuse by hostile applications, a point which is resolved in section 7.8.

7.3 Entirety Bids

Chapter 6 presented in detail the problem of interrelated lots and the consequential combinatorial allocation problem. The solution developed in that chapter featured a combination of RDG and entirety bidding which simplified the allocation problem. Two fine points involving the implementation of entirety bidding are addressed in this section.

There are two versions of entirety bidding described by Milgrom [45]:

1. **Pre-auction:** where the entirety bids are made prior to the piecemeal auction, and the individual lots are only awarded to the winning bidders if their sum exceeds that of the winning entirety bid.

2. **Post-auction:** where the piecemeal auction occurs first.

Due to the single bid nature of the Vickrey auction used in the Nomad Market, all the piecemeal and entirety bids effectively occur simultaneously at time T_c. On receipt of the contract token message the application then determines if it prefers to accept the entirety or piecemeal bids, using its own criteria.

Lastly, each entirety bid over an RDG must include sentences to each and every accept state (lot) in the graph.

7.4 Incorporating Bipartite Negotiations

There may be occasions when only one Depot will be able to satisfy an application's set of constraints, e.g. co-locating with another object, or access to a particular fixed resource. If this information is made public, then the negotiation is open to abuse as the competitive nature of the auction is negated. What is required in order to prevent this abuse is a mechanism preventing the bidder from discovering that it is the only one capable of satisfying the constraints, and therefore the only bidder.

The solution is to introduce a benign form of collusion between the vendor and market. The Depot, that the application needs is known to the application, and this needs to be communicated to the Market and not the target Depot. The application issues an auction with the type set to bipartite and the target Depot is specified in the auction properties (see section 7.7). In addition, the application should construct an RDG with as little Depot identifying information as possible. The most efficient solution is for the market to generate a sanitised catalogue and solicit a bid from the desired Depot. The Depot, unaware that it is the only bidder, will then bid its value. However, a colluding Depot can be used to monitor the catalogues and detect when a bilateral bid is issued (if it registers an identical pattern to the target Depot and does not receive the same catalogue). The only solution to this is for the Market to issue the catalogues to all usual recipients and filter the replies. Sadly this will incur load on some Depots' for an auction in which they have no chance of winning.

Bidders without registered resource descriptions will be sent all catalogues of all lots within the local market, section 7.6.

7.5 Limitations Revisited

With the core of the Market mechanism described, it is reasonable to revisit the eight limitations inherent in the Vickrey auction protocol of: the lying auctioneer, bidder collusion, revelation of sensitive information, lying in non-private value auctions, lower revenue in non-private value auctions, sub-optimal allocation and lying in interrelated auctions, uncertainty of valuation and wasteful counter-speculation. These were detailed in section 5.4.7, and this section examines how the mechanism addresses each one.

To eliminate the problems of the **lying auctioneer** and **revelation of sensitive information** the Market must be trusted. This is done by the making it a Nomad level service. In addition, the Market charges the application a fixed fee (see section 7.8), avoiding the problem of percentage based commissions. On constructing the contract tokens the Market signs the second price bid itself, ensuring that no private value information is released to the application or winning bidder.

Bidder collusion is made difficult through ignorance; a bidder can in no way interrogate the Market and gain information about the other bidders, JIT bidding reduces the time available to plot, and even a single non-collusive bidder renders the agreement ineffective (The collusive bidders have no means of detecting if an outside bidder has bid unlike in the English auction.).

To solve the problems of **Lying in non-Private Value Auctions** and **Lower Revenue in non-Private Value Auctions** contracts are strictly non-transferable. This property is ensured by the mechanism, as the contract and therefore the potentially resalable item does not exist until the application redeems the associated contract token. As the contract token is not binding on the application, and the Depot is not aware if it is the winner before the contract token is redeemed, there is nothing to on-sell. At the time the contract token is redeemed for a contract, the contracted resources are immediately available. It follows, that since contracts are non-transferable, all auctions in the Market are private value.

The provision of entirety bids in the mechanism enables interrelated lots to be allocated optimally, removing the possibility of counterspeculation and lying and resolving the issue of **Sub-optimal Allocation and Lying in Interrelated Auctions**.

This leaves two limitations, the **Uncertainty of Valuation** and consequent **Wasteful Counter-speculation**. Valuations are uncertain when either due to complexity or bounded compute, a bidder cannot compute its true valuation. This is mitigated by two factors in the Nomad system. Firstly, the resource description graph is a directed acyclic graph and finite, and secondly the language of edge expressions is designed for ease of computation and RDGs are expected to be small. Even further, only risk averse bidders are affected by local uncertainty, and as risk is proportional to stake, in the low value Nomad auctions there are likely to be few risk averse bidders.

Finally, assuming there is actually local uncertainty, the proof from [92], that counter-speculation is worthwhile when there is local uncertainty, relies on a set of additional and fragile assumptions. Section 5.4.7 contains a synopsis of the proof, which requires the assumptions: that the valuation v_2 of bidder b_2 is certain, that v_2 is common knowledge for the counterspeculating bidder b_1 and finally that the cost c of obtaining the b_1 valuation v_1 is itself not uncertain. The Market does not supply knowledge of v_2, or in anyway identify b_2 to b_1. Assuming that b_1 is actually aware of v_2, then it must also be aware of the valuations of all other bidders; from b_1's point of view, this problem is equal to bidder collusion and infeasible.

7.6 The Global Marketplace Structure

This chapter has tended to discuss the Marketplace in the singular tense for ease of discussion. However, this is clearly not tenable for a global negotiation service, and the Marketplace itself is a globally distributed application. The Marketplace is composed of a set of service objects (*Markets*), whose clients are divided along regional lines using the same mechanism[1] as the globe quad tree [104], and a coordinating contact object.

The contact object maintains internally a virtual hierarchy, shown as the set of white boxes in figure 7.3, which enables it to determine the local Market for each registering client. When a client (bidder or vendor) registers with the Marketplace, the contact object walks its internal virtual hierarchy, to determine the Market local to the client, and returns an interface on the Market.

[1] The mechanism of dividing the regions, not the quad tree. The Marketplace uses a binary rather than quad structure.

As the load on a Market (the shaded boxes) increases beyond an upper $threshold_\gamma$, it splits, dividing the client load geographically between the two resulting Markets. At the time of the split the dividing Market notifies the contact object, which updates its virtual hierarchy to reflect the new distribution. All the Market's clients are also notified via the `Client handOff` method which they implement.

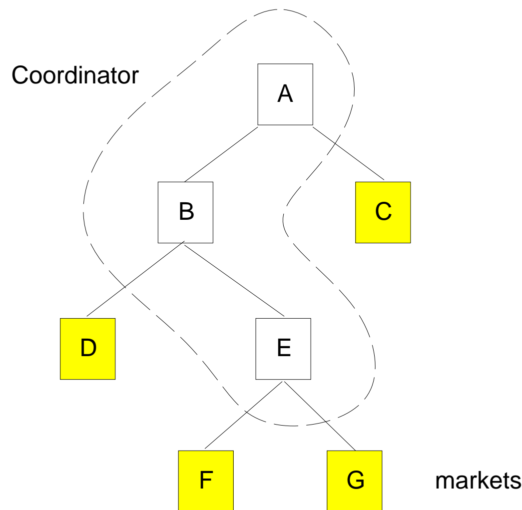


Figure 7.3: *Global Marketplace Structure.*

Likewise, when the load at a Market decreases below a lower $threshold_\mu$ it will merge with its sibling or left or rightmost descendent of its sibling as obviously only adjacent regions may be merged. For example, if in figure 7.3, service object C needs to merge, it will merge with G, leaving D and F untouched. Market D would merge with F, whereas G would merge with F. Again, the contact object virtual hierarchy is updated to reflect the new distribution and clients are notified via the handOff method. To limit thrashing of Markets there is a hysteresis gap between the two thresholds and $threshold_\gamma > threshold_\mu$.

7.6.1 Categories

Categories are hints which are orthogonal to the regional division of the Marketplace. They are simply textual labels which are included in the auction properties and if there is a Market which hosts that category, then the local Market forwards the auction catalogue to the category's Market. Note that the auction is still held by the application's local market, and that the category Market is a mechanism by which the catalogue is distributed

to interested bidders. This means that an auction may have multiple categories, and that even though bidders may register with a category Market, that Market will not as a matter of course host the auction.

If a Market decides to host a particular category, then it is simply becoming a nexus for catalogue distribution, it still services its regional load. Which Markets host which categories is maintained by the co-ordinating Marketplace contact object. If a Market chooses to drop a category, then it is simply removed from the contact object list, no other action is required or taken.

7.6.2 Catalogue Distribution

The means of advertising the individual auctions is through catalogue distribution. Recall that each auction is always held by the application's local Market, but wider distribution of the catalogues permits non-local and category based Depots to bid.

On listing, an auction is assigned a time-to-live (TTL) and is forwarded in a catalogue update to the regional neighbours of the local Market. This is then in turn passed on by those neighbours until the TTL expires. The TTL can be a hop count, or a time computed from the closing time of the auction. Likewise, if any categories are listed, or target regions specified (see 7.7), then these too are sent a catalogue update.

7.6.3 Ordering

Each Market is independent from its peers in the Global Marketplace structure described in section 7.6. Each auction is listed in only one Market, and local Market time is authoritative.

Auctions are distributed globally by exchanging catalogues, however all bids are made to the Market of origin registered in the catalogue, and not the bidder's local Market. Therefore the arrival ordering of bids is resolved internally within in each market and accurate clock synchronisation between Markets is not required. Late bids are simply discarded.

7.7 Auction properties

The auction properties contain information pertinent to the running of the auction itself, rather than the constraint requirements of the applications. At the present time the auction properties contain; closing time, lapse time, type, target region(s)[2] and a textual list of zero or more categories.

The closing and lapse times are specified relative to the local market time, which is authoritative for this auction. The auction type can be either standard, entirety (see section 7.3) or bipartite (see section 7.4).

A target region is usually specified by naming a Depot which is a member of that region; if multiple regions are specified then catalogues are distributed to all of them. As the location service can be used to determine the current Depot host of an object a target region may also be specified by an object. Categories are hints directed to the marketplace by the client and are discussed in detail in the previous section.

7.8 Marketplace Economics

Each Market charges a vendor for each accept state in an auction plus a premium for entirety and bipartite auctions. The former because it is in effect an additional lot, and the latter as more effort is required from the market, and the possibility that an already loaded depot could be being deliberately targeted. This amount covers the costs incurred, and encourages vendors not to indulge in spurious auctions. Bidders are not charged to bid as they already incur costs from evaluating the resource descriptions. The funds collected by the Marketplaces are fully disbursed amongst its hosts to offset its residence. That is, the Nomad system does not profit from the transactions (lying auctioneer).

Since a contract token is binding on the bidder but not the vendor, it is possible that the vendor may choose not to accept the winning bid. Thus the contract token only becomes a formal binding contract once it has been redeemed for service.

To remove the spectre of a lying auctioneer, all auctions have a fixed fee *plus a premium per accept state*. This should minimise the abuse by and of the market place and reduce

[2] If the type is bipartite, the the target refers to a specific Depot, otherwise the target defines the Market region of which the specified Depot is a member.

the number of unnecessary bids generated by Depots. Due to the low cost of negotiation, the system overhead should not be great. Bidders are not charged, as they spend resources evaluating auction resource descriptions.

Being a Nomad system level application, the Marketplace is not intended to be profitable, and any funds generated are used to compensate depots hosting the components of a Marketplace. The Marketplace does not require a contract with its host depot and is not charged any service fees. A Depot cannot opt out of hosting a Nomad system level application and must carry this burden in exchange for belonging to the system. The marketplace will in turn, pass on fees to hosting depots and attempt to distribute the load of its execution fairly over the applicable set of trusted Depots.

7.9 Fault Model

Globally optimal resource allocation cannot be made in real-time, as although optimal allocation algorithms exist [65, 76] they involve not one but multiple *NP*-complete computations.

The Nomad negotiation mechanism does not seek globally optimal allocations, rather it divides the vendors and bidders into regions and then makes an approximate allocation within this region. This is a compromise between optimality and tractability, and is additionally justified by the low value of the resources being negotiated. The approximation removes the multiple *NP*-complete computations from the regional Market and places one, in the form of an RDG, in the hands of each bidding Depot. The Depots compute their bids in parallel giving a wide distribution to the allocation computation.

By restricting the possible bid combinations to only those which are valid (specified by the vendor in the coding of the RDG), the Depot is faced with a very much better defined problem. One that is still, however, *NP*-complete, particularly if multiple RDGs (auctions) are evaluated at one time. This emphasises two problems. Firstly, the size of the RDGs cannot be too large or the load on each Depot will quickly become excessive. Secondly, the edge expressions which compose the RDG must be carefully specified to minimise computation and inclusion of further *NP*-complete problems. Both of these problems form routes of attack on the negotiation system and must be rigorously policed by both the Markets and the Depots themselves.

The minor *NP*-complete problem at each Depot can be entirely eliminated by further restricting the Depot to:

- making either just a single lot bid or an entirety bid for each auction, and
- only bidding on one auction at a time.

These restrictions are excessive, however. Providing that the size and edge specifications are enforced, the bid computation will be sufficiently tractable.

7.10 Summary

The Nomad negotiation service is a distributed resource allocation system, which enables applications to negotiate and pay for execution resources. The mechanism is simple, allowing computationally bound clients to construct resource descriptions from building blocks, and to negotiate simply and efficiently.

The Vickrey auction is an ideal basis for automation, provided that its limitations are addressed in the design of the protocol mechanism. The distribution is Pareto-optimal, proofs of which are available in Rasmusen [87] amongst others, and consequently also maximises social welfare. The protocol satisfies the requirement for stability, by having a dominant strategy, while the system is individually rational as truthful bidding ensures that the payoff is not less than not participating. That is, even if the second bid is equal to the first, no loss will be made. Lastly the system is symmetric, as the same behaviours will result in the same payoffs.

The mechanism design in the Nomad Marketplace addresses each of the limitations, and therefore satisfies the negotiation metrics from section 5.2, within the constraints of tractability from section 7.9. Coordination within the Market is via the distribution of catalogues, and categories provide a means of advertising for specific types of Depots. Bipartite auctions are simulated within the multipartite auction protocol, in such a way that the Depot will bid its true value.

The combination of Vickrey auction, resource description graph and Market mechanism, provides a tractable and novel solution to the problem of distributed resource allocation.

Chapter 8

Experimental Prototype

This chapter documents a proof-of-concept implementation of, and experiments on, the Nomad architecture. There are three important aspects of the Nomad architecture which require experimental validation. Firstly, the location service utilises mobile location tables within each application to track and resolve that application's objects. This presents the underlying mechanism with the problem of trying to resolve one out-of-date reference with another, that may itself be out-of-date. Section 8.2 describes a test application that verifies this mechanism in the prototype, demonstrating this technique can be successfully utilised. Secondly, the Nomad negotiation service provides a means for both applications and Depots to express their resource policies and therefore both influence the outcome of resource negotiation. Section 8.3 documents a set of experiments carried out on the Nomad prototype which verify this. Lastly, the high level Nomad infrastructure is intended to make the creation of distributed applications easier. Section 8.4 presents the design and straightforward implementation of two such distributed Nomad applications.

This chapter starts first with a contextual description of the prototype itself.

8.1 Prototype

The prototype is a proof-of-concept implementation of the Nomad architecture. The prototype includes functional implementations of: Depots and their managers, location service, negotiation service and applications to inhabit and test the Nomad environment. These components all act as described in their relevant chapters, and where simplification

has occurred it is in the implementation rather than in function. The only significant component missing from the prototype is a billing system, which is unnecessary in this proof-of-concept.

8.1.1 Implementation

The prototype is implemented in the Java programming language, and is built on top of Flexinet [40] utilising its remote method invocation, generic proxies, and mobility packages.

Flexinet is itself a prototype rather than production system and does not handle host or communications failures, and required extensive debugging during construction of the Nomad prototype. Flexinet also suffers from poor performance, poor resource utilisation and general fragility. This aside, Flexinet was an ideal platform, sharing many of the design principles and permitting extension of its components. Certainly, without it, the prototype would not have been built.

The Yellow Pages and location table locator are implemented as centralised client-servers rather than as distributed applications, and the prototype does not yet support persistence or replication. None of these points reduce the value of the prototype as a proof-of-concept.

To start an application from outside Nomad, a special application loader is used called the Launcher. The Launcher negotiates for any resources required and then in turn creates the new application.

8.2 Location Service

One of the features of the location service is that part of the location service, the location tables, are mobile themselves. The purpose of this section is to demonstrate that such a mechanism is feasible, and this requires a working example of mobile location tables.

The ideal application for this is Tweetie Pie, which is taken from the Flexinet [40] test suite. This example application works as follows. The Tweetie Pie Launcher negotiates for two vHosts, one which we will refer to as the `cage` and the other as the `room`. These are both passed to the Tweetie Pie application when it is created in the `cage`. Tweetie Pie implements an external interface `Talk` which is now returned to the Launcher.

After some time interval τ in the `cage` Tweetie Pie sees Sylvester and flies to the `room`. This process then repeats after another τ, with Tweetie Pie flying from the `room` to the `cage`. After some time interval υ, the Launcher invokes the `say` method on Tweetie's `Talk` interface which makes Tweetie Pie print a message.

To fully test the rebinding of mobile location tables, Tweetie Pie is modified to move its location table as well as itself. When Tweetie Pie flies between the `cage` and `room` or vice-versa, it first invokes the `move(dest)` method on its location table, causing the location table to move first to `dest`.

Consider what location resolutions now occur within the system. At the beginning both Tweetie and the location table are started in the `cage`. At this time, they both hold correct references to each other, and the Launcher holds an accurate Talk interface on Tweetie. After time interval τ, Tweetie invokes the move method on its location table to move it to the `room`. During the move the location table automatically updates the location table locator with its new location. Next Tweetie also moves to the `room`. As the location table moved since Tweetie last referenced it, when Tweetie re-maps its new location the location table reference is out-of-date. This reference is then transparently rebound via the location table locator, and the remap on the location table performed, as shown in figure 4.7 on page 49. Tweetie now holds the correct reference, and they are both located in the `room`. Assume the Launcher is now at interval υ, and it invokes the `say` method on Tweetie's `Talk` interface. This reference is out-of-date and points to the `cage` rather than the `room`. Resolving this requires that first the location table be rebound via the location table locator, and then Tweetie rebound via the location table in a cascade of rebinding, as shown in figure 4.8 on page 50.

This functioning application successfully demonstrates that the mobile location tables rebind transparently both internally within the application (update during mobility) and for external clients (invoking `say`).

8.3 Negotiation System

The Nomad architecture is a framework, and the resource policies used in the management of a system based on this framework are separate and not directly part of the mechanisms developed in this thesis. However, some basic policies are needed to demonstrate that

the resource negotiation mechanism performs as expected. The policies, on which the experiments in this section are based, are designed to show that the negotiation service allows applications to obtain the resources they require, and Depots to control their loads. That is, that both applications and Depots can influence distribution.

The experiments selected for this section apply different arbitrary policies to demonstrate that both applications and Depots influence distribution. It is not the quality of the policies that is under investigation in this experimentation, rather, the significance is in the response of the system to policy.

Section 8.3.1 describes the Tribble application designed for these experiments, and section 8.3.2 describes the experimental Nomad platform and details the exact configuration of the system and Tribbles. The experimental results are presented in sections 8.3.3 through 8.3.6.

8.3.1 Tribbles

The application designed to exercise the negotiation system is the Tribble. It is a resource consumer designed to consume real resources such as CPU, memory and IO over a time interval τ, and monitor that resource usage. The system starts with one Tribble, after τ seconds the Tribble negotiates for two vHosts, one on which it creates a new Tribble, and the other to which it moves. This cycle is illustrated in figure 8.1.

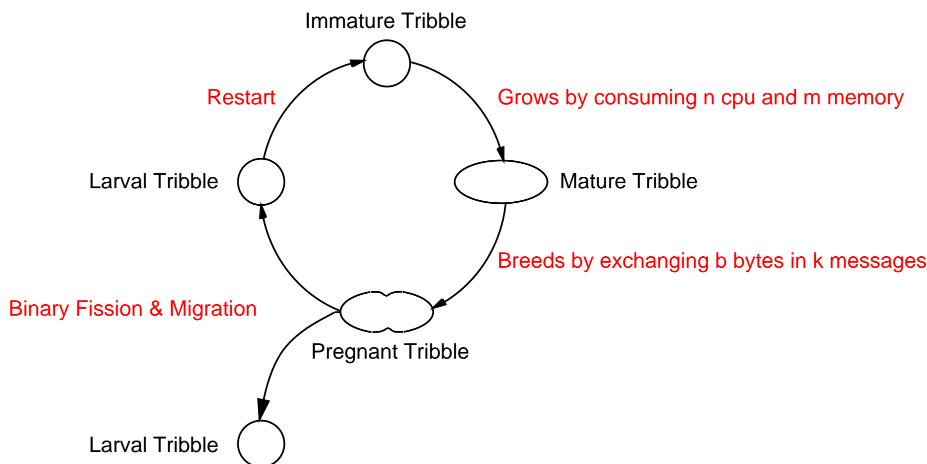


Figure 8.1: *Life-cycle of a Tribble.*

The life of a Tribble begins when it is created or migrated on to a vHost in its larval state.

The Tribble then starts or restarts, creates its transient execution state, and advances to its immature state. Before the Tribble reaches maturity, it consumes its designated resources over time τ. Once a Tribble is mature it breeds by contacting a random Tribble, selected from its Tribble list, and sharing its Tribble list with it[1]. Following breeding, the Tribble negotiates for two new vHosts and creates a child Tribble on one. The Tribble then migrates onto the second vHost and reverts to the larval state.

To control the Tribble population, Tribbles have a generation limit which restricts the population to a specified power of two. Once the limit is reached, Tribbles continue to negotiate and move after τ, but no longer undergo binary fission.

8.3.2 Experimental Platform

The experimental setup for evaluating the negotiation system is based around a set of nine machines. These form a three by three grid with a Depot at each xy grid intersection. There is a full location service, with the Yellow Pages and location table locator resident on the Depot at $(2, 2)$. Likewise, there is a single market which executes on the Depot at $(3, 3)$.

All nine Depots were executed on Pentium III-500 netBSD machines on the same subnet. As the experiment used homogeneous hosts and homogeneous test applications (Tribbles), the load metric was simplified by directly counting the number of Tribbles on each Depot. These Tribble counts are logged from each Depot at two second intervals. If this coincides with a Tribble moving, then it may be counted at both the source and destination. This is why all of the result matrices in the following experiments have a sum very close to nine rather than eight. This is reasonable as load is incurred on both Depots during migration.

The Tribble application is limited to a population of eight. The Tribble life-cycle interval τ is twenty seconds, and each auction defaults to a ten second period with a fifteen second lapse time.

The limitations of Flexinet, as outlined in section 8.1.1, preclude the use of larger grids or more Tribbles. Specifically a currently untraceable memory leak in the Flexinet object serialisation code cripples the market when using greater numbers of Tribbles or Depots.

[1] This list exchange tends towards complete knowledge of all Tribbles descended from the initial Tribble.

However, the results obtained are accurate and sufficient to demonstrate the effect of negotiation policies on distribution.

8.3.3 Random Valuation

In the following experiments, Tribble resource requirements and therefore their policies are denoted by α. Depot negotiation policy and therefore bid valuations are specified with β.

For the random valuation experiment, Tribbles have no resource requirements ($\alpha = none$) and the RDG is simply the sentence $\{true\}$ to a single accept state. The Depot policy (β) is to generate random bids, unrelated to their current load. This experiment forms the baseline against which comparisons of later experiments can be made. The result matrix in table 8.1 shows, for each Depot at grid position xy, the average number of resident Tribbles ($\bar{n}_{xy}$) and standard error[2] ($\sigma_{\bar{n}}$) defining a 95% confidence interval.

$$\alpha = none, \quad \beta = random$$

$$\bar{n}_{xy} \pm \sigma_{\bar{n}} = \begin{bmatrix} 0.80 \pm 0.06 & 1.01 \pm 0.06 & 0.80 \pm 0.07 \\ 0.95 \pm 0.07 & 1.28 \pm 0.07 & 0.95 \pm 0.07 \\ 0.83 \pm 0.06 & 0.96 \pm 0.07 & 1.52 \pm 0.08 \end{bmatrix}$$

Table 8.1: Random Depot selection.

The intervals from all grid positions except $(2,2)$ and $(3,3)$ overlap with at least one other grid position, indicating a general lack of statistical significance between the means. This is entirely reasonable considering the random bidding policy. The Depot at (3,3) has a significantly higher mean and hosts the market. As tied bids are broken on arrival order, and this Depot has a shorter communication delay with the market, it can be expected to be favoured in ties. Ties are not uncommon as random bids fall within the range of $1 \rightarrow 10$).

[2] A 95% confidence interval by the Central Limit Theorem is: $\sigma_{\bar{x}} = 1.96\frac{s}{\sqrt{n}}$

8.3.4 Uniform Cost Model

This experiment has a uniform cost model for all the Depots. Each Depot uses the same policy β, which is the number of Tribbles currently resident on that Depot. The Tribbles are identical to those of the random valuation experiment. This is equivalent to a policy of least cost due to the marketplace auction protocol. The assumption behind this experiment was that the Tribbles would evenly distribute themselves over the entire grid.

$$\alpha = none, \quad \beta = n$$

$$\bar{n}_{xy} \pm \sigma_{\bar{n}} = \begin{bmatrix} 0.93 \pm 0.07 & 0.74 \pm 0.06 & 1.13 \pm 0.09 \\ 1.05 \pm 0.07 & 0.70 \pm 0.05 & 0.82 \pm 0.06 \\ 1.24 \pm 0.09 & 0.56 \pm 0.05 & 1.78 \pm 0.12 \end{bmatrix}$$

Table 8.2: Uniform Depot selection.

However, as can be seen from the result matrix in table 8.2, the averages in the uniform cost model are less similar than random. This is due to a lack of bidding sophistication by the Depots. There is a long delay between making a bid at τ, and the arrival of the Tribble on the Depot at v. All bids from this Depot between τ and v are therefore made with an out-of-date Tribble count. Since this Depot has won the first auction at time τ, it could win all subsequent auctions until time v, unless another Depot bids less due to an existing Tribble leaving before v. In effect, this results in cyclic swamping of one Depot after another, and it is therefore not surprising that the results are worse than random. This agrees with the results in [18], where simple-minded least-loaded distribution policies need some form of anti-swamping algorithm to outperform random.

This data has fewer overlapping means reflecting by the increased significance, greater underlying structure. This however, does not mean better load distribution. A small number of sub-experiments were run to see if these results could be improved without using a sophisticated resource reservation scheme. The first was to reduce the auction period (t_c) from ten seconds to three, and the lapse time (t_l) from fifteen seconds to five. These results are shown in table 8.3.

Also tried was de-committing, that is, once a contract is redeemed at Depot A, all outstanding bids by A are withdrawn. The results for this sub-experiment are shown in

$$\alpha = none, \quad \beta = n, \quad t_c = 3, \quad t_l = 5$$

$$\bar{n}_{xy} \pm \sigma_{\bar{n}} = \begin{bmatrix} 0.86 \pm 0.06 & 0.56 \pm 0.04 & 0.81 \pm 0.05 \\ 0.94 \pm 0.07 & 0.70 \pm 0.04 & 0.61 \pm 0.05 \\ 1.07 \pm 0.08 & 1.78 \pm 0.10 & 1.81 \pm 0.10 \end{bmatrix}$$

Table 8.3: Shorter auctions.

table 8.4. In practice neither modification improved the results and therefore a more sophisticated bidding policy with an anti-swamping algorithm, such as that from [18], is needed. These results, however, significantly and consistently confirm that the Depot hosting the market is favoured in the uniform bidding auctions.

$$\alpha = none, \quad \beta = n, \text{ with bid retraction}$$

$$\bar{n}_{xy} \pm \sigma_{\bar{n}} = \begin{bmatrix} 1.10 \pm 0.08 & 0.58 \pm 0.05 & 1.23 \pm 0.09 \\ 0.74 \pm 0.06 & 0.70 \pm 0.06 & 0.81 \pm 0.07 \\ 1.15 \pm 0.09 & 0.55 \pm 0.06 & 2.24 \pm 0.14 \end{bmatrix}$$

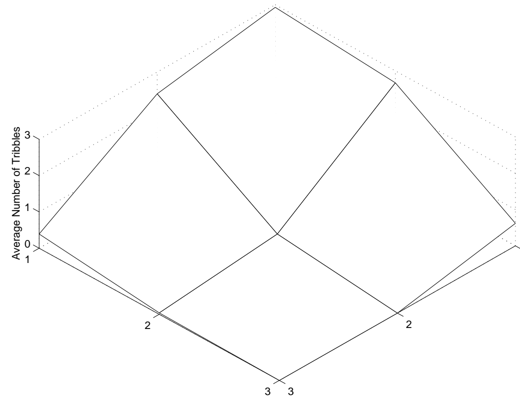
Table 8.4: Depot bid retractions.

8.3.5 Nonuniform Cost Model

The goal of the nonuniform cost model experiment is to demonstrate the impact of Depot policy on the distribution of Tribbles. The Depot cost model β is the sum of the Depot's x and y grid components over two, plus the number of Tribbles on the Depot. Table 8.5 shows the Tribbles clustering on the Depots with low grid references and therefore correspondingly low bids. Note that the graph is rotated so that Depot $(3, 3)$ is at the front.

One interesting point revealed in table 8.5 is that the differences in the means in the diagonals (e.g. [0,1][1,0]) are not statistically significant (e.g. the 2.67 and 2.40 95% confidence intervals overlap). This is a good outcome, as these grid locations have the same sum, and therefore the same weighting in bidding. This is demonstrated to an even clearer extent in table 8.6. These show the results for a similar experiment, but with a greater weighting placed on the grid values that the load.

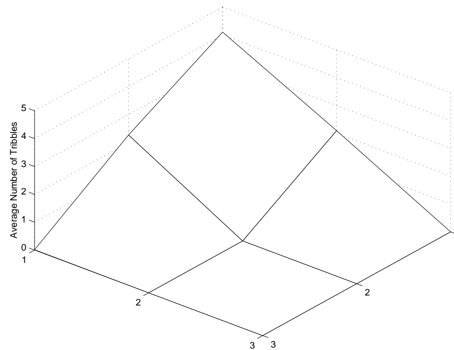
$$\alpha = none, \quad \beta = (x+y)+n$$



$$\bar{n}_{xy} \pm \sigma_{\bar{n}} = \begin{bmatrix} 2.93 \pm 0.18 & 2.40 \pm 0.17 & 0.39 \pm 0.06 \\ 2.67 \pm 0.18 & 0.36 \pm 0.05 & 0.04 \pm 0.01 \\ 0.62 \pm 0.07 & 0.00 \pm 0.00 & 0.00 \pm 0.00 \end{bmatrix}$$

Table 8.5: Nonuniform cost model.

$$\alpha = none, \quad \beta = 3(x+y)+n$$

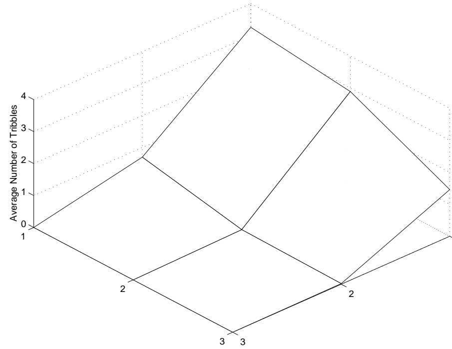


$$\bar{n}_{xy} \pm \sigma_{\bar{n}} = \begin{bmatrix} 4.10 \pm 0.17 & 2.10 \pm 0.13 & 0.00 \pm 0.00 \\ 2.27 \pm 0.13 & 0.00 \pm 0.00 & 0.00 \pm 0.00 \\ 0.01 \pm 0.01 & 0.00 \pm 0.00 & 0.00 \pm 0.00 \end{bmatrix}$$

Table 8.6: Nonuniform cost with greater grid weighting.

The use of the sum of the grid reference is arbitrary and unrelated to any real constraint, adjusting the weighting to produce a more interesting distribution is reasonable and further demonstrates the impact of Depot policy. Table 8.7 shows the effect of changing the weighting on one of the grid components. The resulting data and graph illustrate this nicely with a clear preference exhibited for one side of the grid.

$$\alpha = none, \quad \beta = (2x + y) + n$$



$$\bar{n}_{xy} \pm \sigma_{\bar{n}} = \begin{bmatrix} 3.26 \pm 0.17 & 0.70 \pm 0.07 & 0.00 \pm 0.00 \\ 2.98 \pm 0.17 & 0.07 \pm 0.02 & 0.00 \pm 0.00 \\ 1.46 \pm 0.11 & 0.02 \pm 0.01 & 0.00 \pm 0.00 \end{bmatrix}$$

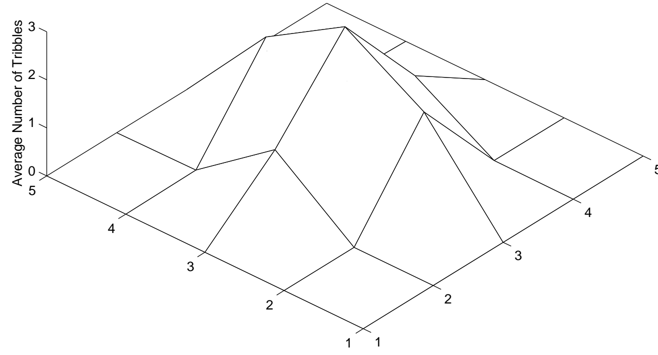
Table 8.7: Nonuniform cost with greater grid weighting on x.

8.3.6 Tribble Constraint

The final experiment introduces a Tribble constraint to establish the effect of Tribble policy on distribution. In this scenario the Tribbles express a preference for the Depot at the centre of the grid. This results in a resource description graph with the root transition expression α. Depots use a uniform cost model based on the number of Tribbles currently hosted and the Euclidian distance from the current Depot (x', y') to the preferred Depot (x, y) .

The results are given in table 8.8. The clear peak in the middle is a positive confirmation that Tribble preferences encoded in the resource description graph influence distribution. This is a statistically significant result as none of the confidence intervals overlap grid positions with different sums. The graph has been extended with an edge of zeros to make the visualisation clearer, these do not represent actual data points, rather expected values.

$$\alpha = near(host_{xy}), \quad \beta = n + \sqrt{(x - x')^2 + (y - y')^2}$$



$$\bar{n}_{xy} \pm \sigma_{\bar{n}} = \begin{bmatrix} 0.00 \pm 0.00 & 1.92 \pm 0.13 & 0.01 \pm 0.01 \\ 1.24 \pm 0.10 & 2.90 \pm 0.18 & 0.98 \pm 0.09 \\ 0.02 \pm 0.01 & 1.89 \pm 0.14 & 0.00 \pm 0.02 \end{bmatrix}$$

Table 8.8: Tribble Constraints showing Tribble policy influencing distribution.

8.4 Whiteboard

One of the primary goals behind the Nomad architecture is high level support for the distributed application programmer. The best way to illustrate this is by presenting a small application which deals with distribution. An application that has been used effectively in this demonstrative role is the whiteboard [62]. A whiteboard application typically provides shared whiteboards on which several users can write or draw simultaneously, with each seeing an up-to-date representation of the complete whiteboard state. There are two approaches:

- a standard client-server model utilising RPC or remote method invocation, and
- a distributed application model with components of the application residing on different machines.

This section will discuss both of these approaches to the white board application, as it is the differences between the two that best illuminate features of the Nomad programming environment. It is important to note that this is not an exercise in how to build a good

whiteboard application, rather it is purely a vehicle for illustrating Nomad. Thus, the whiteboard is as simple as possible, yet still captures the important aspects of the distribution problem. Many additional features could be incorporated, such as recording state, to bring new users up-to-date, drawing sketches, group communication, multi-casting, or indeed, more than one whiteboard. However desirable, these are irrelevant to the demonstration at hand. This test whiteboard simply exchanges strings amongst members of a single whiteboard and retains no record of any previous state.

8.4.1 The Client-Server Model

The popularity of the client-server paradigm in modern systems is due in part to the simplicity of the model, and in part due to the lack of distributed object systems. The basic structure of the application is based on one server, with a single whiteboard and many clients. The server structure is detailed in figures 8.2 and 8.3 and consists of a server program and local data-structures, of which the most important is the *client list*. The client list records the subscribers to the shared whiteboard by storing their interfaces.

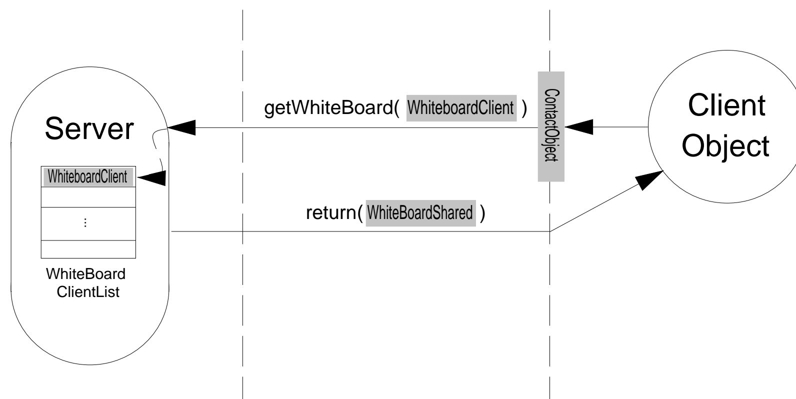


Figure 8.2: *The client-server model of the whiteboard application.*

Each whiteboard user runs a copy of the whiteboard client application, which establishes a connection with the whiteboard contact object. The client then subscribes to the shared whiteboard by invoking the `getWhiteBoard()` method, passing as an argument its own `WhiteBoardClient` interface. This interface is then added by the server to its client list, and the server's interface `WhiteBoardShared` is returned to the client.

When a user writes to the whiteboard, the client does not first modify the local copy of the whiteboard, but sends the changes directly to the server by invoking the `write` method on

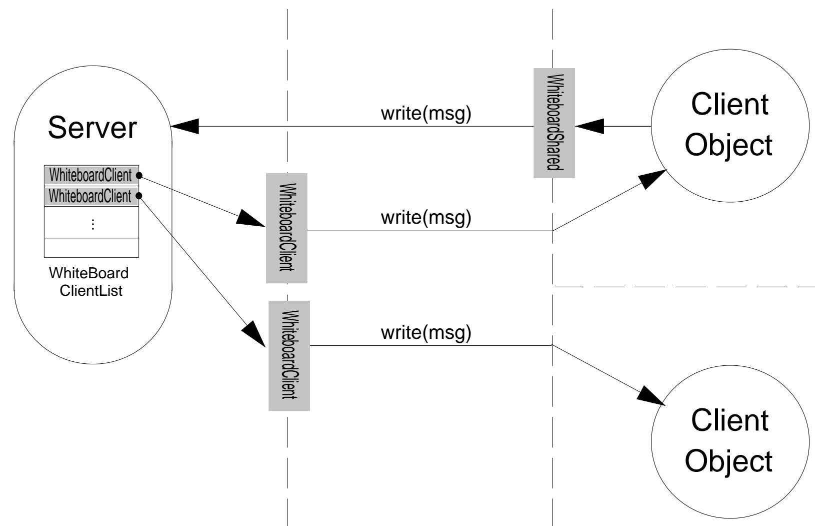


Figure 8.3: *Writing in the client-server model of the whiteboard application.*

the `WhiteBoardShared` interface. The server responds and updates the global whiteboard by propagating the changes to all clients in the client list via the `write` method on their stored `WhiteBoardClient` interface. Another simplifying assumption is that the whiteboard does not require causal ordering of updates [62]. Where two users simultaneously update their whiteboards the local copies may differ.

Interfaces

Three basic interfaces are common to both the client-server and distributed implementations of the whiteboard application. The implementation of the whiteboard client is also common to both and is included in appendix B.1.

```
public interface ContactObject{
  public WhiteBoardShared getWhiteBoard(WhiteBoardClient);
}

public interface WhiteBoardClient{
  public void write(String);
}

public interface WhiteBoardShared{
  public void write(String);
}
```

When a client joins the shared whiteboard service, it must invoke the `getWhiteBoard` on the whiteboard contact object, and provide its implementation of `WhiteBoardClient` as

the argument. The `getWhiteBoard` method returns the `WhiteBoardShared` interface on the shared whiteboard.

Essentially the `WhiteBoardShared` and `WhiteBoardClient` interfaces are symmetric, the former allows the client to write to the shared whiteboard, and the latter enables the WBS to write to the client's local copy of the whiteboard. All clients using the WBS must therefore implement the `WhiteBoardClient` interface.

Implementation

It is now worth examining the actual implementation of the client-server whiteboard. The application's initial task is to register a name for the WBS, otherwise the application is as described in section 8.4.1

```
imports ...;

public class ContactObjectClientServer extends AuthMobileObject
  implements ContactObject, WhiteBoardShared{

  public Vector clients;
  public String name = "";
```

The most significant point in this declaration are the two interfaces it implements. This means that an object of this `ContactObjectClientServer` class fulfils both the role of contact object and shared whiteboard. Consequently, the application does not have to deal with the distribution of any of its objects, except for the location table, as all the other application objects exist within this one cluster. Indeed it does not even deal with its own placement, that function is performed by the Launcher application.

```
  public ContactObject init(String name){
    // Newly created, need to create YellowPages entry.
    clients = new Vector();
    this.name = name;
    register();
    return (ContactObject)this;
  }

  public void restart(Exception e){
    // Must have moved, update YellowPages entry.
    register();
  }
```

The `init` and `restart` methods are required by the Flexinet mobility code, and are used to create and restart mobile objects. In both cases the contact object instance invokes the

`register` method.

```
private void register(){
  IfaceRepository lir = ((VHost)getPlace()).getLocalInterfaceRepository();
  IfaceRepository yp = (IfaceRepository)lir.get("/nomad/services/YellowPages");
  yp.put("/nomad/applications/whiteboard", this, ContactObject.class);
}
```

This method registers the WBS with the YellowPages, as is described in section 4.3.4. The first step is to get the LIR from the vHost, and using that acquire an interface on the YellowPages. Finally the YellowPages creates a mapping between the service name `/nomad/applications/whiteboard` and the contact object (this). The URN prefix `urn:nomad:` is assumed by the YellowPages implementation.

```
  // ContactObject method

  public WhiteBoardShared getWhiteBoard(WhiteBoardClient wbc){
    clients.addElement(wbc);
    return(this);
  }
```

The `getWhiteBoard` method is the implementation of the contact object interface. This method responds to an invocation by adding the passed `WhiteBoardClient` interface to clients (client list), and returns a `WhiteBoardShared` interface on itself.

```
  // WhiteBoardShared method

  public void write(String s){
   Enumeration e = clients.elements();
   while(e.hasMoreElements()){
     WhiteBoardClient wbc = (WhiteBoardClient)e.nextElement();
     wbc.write(s);
   }
 }
}
```

The final method in the WBS is the `write` method. This is invoked by a client update and responds by sending updates to each and every registered whiteboard client via their interfaces stored in the `clients` Vector. This application, including the interfaces and test client, required less than one hour and one page of code to implement.

8.4.2 The Distributed Application Model

The client-server model while simple to implement in Nomad, fails to take full advantage of the power of the distribution system. The alternative distributed structure of the white

board application uses replicated service objects placed near to the client, as illustrated in figures 8.4 and 8.5, which maintain copies of the shared whiteboard. In this model the service and contact objects may or may not reside on the same Depot. On the upper left of figure 8.4 is the white board contact object, which is a close analogue to that in the client-server model, but with references to service objects rather than clients.

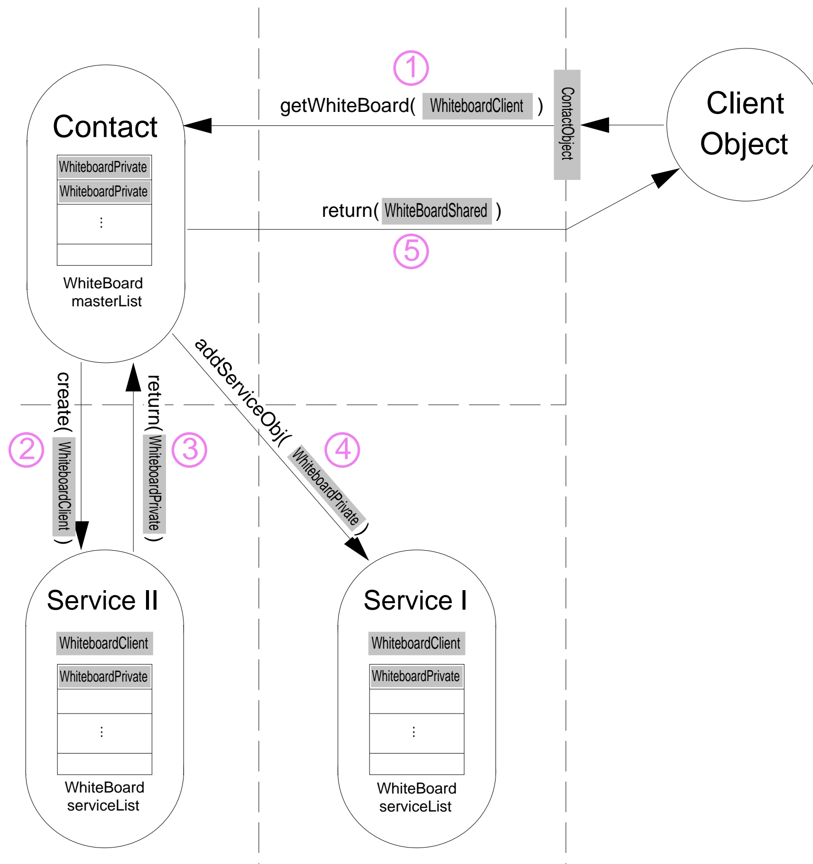


Figure 8.4: *Registering with the distributed whiteboard application.*

From the point of view of the client, its interaction with the contact object is identical to that in the client-server version, as is its interaction with the shared white board. However, what goes on within the contact object and its application is very different. When the client invokes the `getWhiteBoard` method ① on the contact object, the contact object responds by firstly negotiating for a vHost near the client, and then creating a new service object. On creation, ② the service object is passed the `WhiteBoardClient` interface, for this client which it will service. Returned to the contact object ③ is a `WhiteBoardPrivate` interface, used by the contact object to control the service object,

obviously this interface is not passed to a client. Next, ④ the contact object passes the new interface to all other service objects, and then records the interface in its master list. Finally, the contact object obtains a `WhiteBoardShared` interface from the service object, and returns ⑤ this to the client. The `WhiteBoardShared` interface now represents a service object rather than a contact object. After the initial negotiation with client and creation of new application structure, the contact object has no further role in the maintenance of the shared whiteboard.

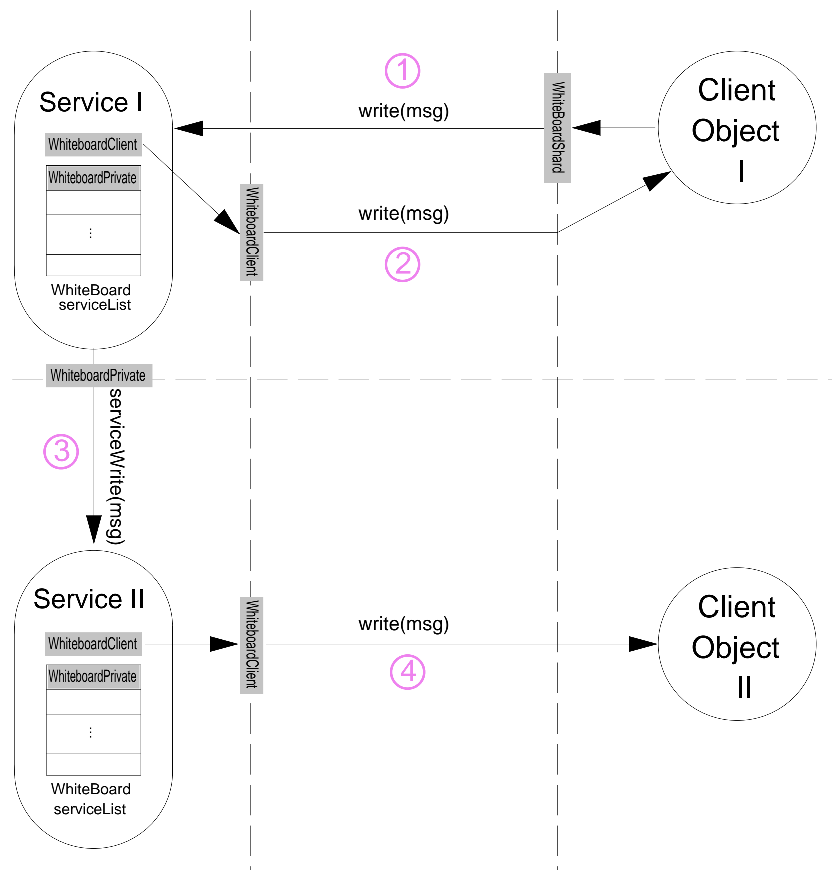


Figure 8.5: *Writing in the distributed whiteboard application.*

When the user writes to the WBS, as illustrated in figure 8.5, it invokes the `write` method ① on the `WhiteBoardShared` interface on the service object. The service object updates its client ② and then all other peers ③ using their interfaces from the `serviceList`. Each service object then updates its associated client ④.

Implementation

The distributed implementation requires additional sophistication to deal with the distribution of its constituent parts. This includes one additional interface, the service object implementation, and the interaction with the Nomad distribution and negotiation systems. The complexity is entirely associated with creating the local `WhiteBoardPrivate` instance *near* the associated client. This exercises the entire Market mechanism. The additional `WhiteBoardPrivate` interface, shown below, is used by the contact object to control the service objects, and for service objects to communicate between themselves.

```
public interface WhiteBoardPrivate{
  public void serviceWrite(String s);
  public void addServiceObj(WhiteBoardPrivate wbp);
  public WhiteBoardShared getPublicInterface();
}
```

There are no changes to the three interfaces used previously in the client-server application. The client application remains unchanged and the implementation of the new service object is given in appendix B.2. The changes to the contact object are extensive, and are presented below.

```
imports ...;

public class ContactObjectDistributed extends AuthMobileObject
  implements ContactObject, Vendor{

  public Hashtable auctions;  // list of the auctions
  public Vector whiteboards;  // Master list of whiteboards
  public Vending vending = null;
  public String name = "";
```

The distributed WBS contact object no longer implements the `WhiteBoardShared` interface, this is now implemented by the service object. Instead the contact object implements the `Vendor` interface which is required by all applications using the Marketplace for negotiation. Other differences include the `auctions` hashtable, used to match auction results to each auction created, and the `whiteboards` Vector, which is the master list of service objects.

```
public ContactObject init(String name){
  auctions = new Hashtable();
  whiteboards = new Vector();
  this.name = name;
  register();
  return (ContactObject)this;
}

public void restart(Exception e){
  // Must have moved, update YP entry.
  register();
}

private void register(){
  IfaceRepository lir=((VHost)getPlace()).getLocalInterfaceRepository();
  IfaceRepository yp=(IfaceRepository)lir.get("/nomad/services/yellowPages");
  yp.put("/nomad/applications/whiteboard", this, ContactObject.class);
}
```

Again, the `init` and `restart` methods are required by the Flexinet mobility code. The only change from the client-server version is the initialisation of the new data-structures. The `register` method is unchanged.

```
private String getRDG(WhiteBoardClient wbc){
  FlexiStub asStub=(FlexiStub)wbc;
  String addr=(((MobileName)asStub.getName()).cluster.address.ref).toString();
  String host=addr.substring(0,addr.indexOf(":"));
  return new String("([near("host+”)](a true (())))”);+
}
```

The `getRDG` method constructs the resource description graph shown in figure 8.6 for use in negotiations. The root transition expression is `near(host)`, which is the only application constraint.

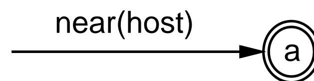


Figure 8.6: *The resource description graph.*

The majority of the code in this method extracts the client's physical location from its `WhiteBoardClient` interface. It would be easier to pass a reference to any object (host or cluster) to the Marketplace, however, this would require the market to either pass this work on to each bidder — and therefore result in the extraction being repeated many times, or have the Marketplace alter the resource description graph. In both cases this would require the market or bidders to locate, load and resolve the `WhiteBoardClient` interface. This is inefficient and expensive, so such extractions are always performed by

the market client.

```
public WhiteBoardShared getWhiteBoard(WhiteBoardClient wbc){
  if(vending == null){
    IfaceRepository lir=((VHost)getPlace()).getLocalInterfaceRepository();
    IfaceRepository yp=(IfaceRepository)lir.get("/nomad/services/yellowPages");
    Contact market=(Contact)yp.get("/nomad/services/marketPlace");
    vending = market.amVendor((Vendor)this);
  }
```

The `getWhiteBoard` method is where most of the differences between the client-server and distributed versions of the whiteboard lie. Due to its length, this method has been divided into four segments which reflect logical stages. The first segment above uses the `LIR` and then the YellowPages to obtain a reference on the Marketplace's contact object. The `amVendor` method is then invoked to register with the market. The application can now use the returned `Vending` interface to create and control a resource auction.

```
  // Negotiate for execution resources near the client.

  Results r = null;
  do{
    Object o = new Object();
    LotReference lr = vending.placeAuction(this,
                                           getRDG(wbc),
                                           new AuctionProperties());
    synchronized(auctions){
      do{
        auctions.put(lr, o);  // because can't store null in a hashtable
        try{
          auctions.wait();
        }catch(java.lang.InterruptedException e){}
        o = auctions.remove(lr);
      }while(!(o instanceof Results));
    }
    r = (Results)o;
  }while(r.size() < 1);
```

This segment carries out the negotiation for a new vHost. The outer loop creates the auction, and then tests if the corresponding result is satisfactory. If not, the auction is repeated. The inner loop waits on the auction result from the market, and is woken each time a result for this application arrives via the `resultNotification` method. If it is not the result for the auction that this thread is waiting on, then the loop resumes the wait. This use of the `auctions` hashtable allows the contact object to conduct simultaneous auctions for multiple whiteboard clients. The code needed to conduct a single auction is really just the `placeAuction` invocation, and the `resultNotification` method.

```
    // Process the auction results and create a new Service object

    ContractToken ct = ((LotResult)r.firstElement()).getContract();
    DepotContact depot = ct.getService();
    VHost vhost = depot.getVHost(ct);
    Authority auth = new Authority();
    auth.addCertificate(new Certificate(String.valueOf(ct.getCertificate())));
    WhiteBoardPrivate wbp = null;
    Object[] args = new Object[1];
    args[0] = wbc;
    try{
      Cluster cluster = vhost.createHintedMobileAuthCluster(auth,
                                                       getLocationTable(),
                                                       null);
      wbp = (WhiteBoardPrivate)cluster.createObject(WhiteBoardService.class,
                                                args);
    }catch (Exception e){
      System.out.println("Failed to create WhiteBoard: "  e); +
    }
```

Once a valid auction result is returned, the winning Depot's contact object is extracted from the `ContractToken` and used to obtain a vHost. The next step is to create an application cluster on the vHost, using the certificate generated by the market. This certificate ensures that only the holder of the `ContractToken` may use the vHost. The final step is the creation of the service object within the new cluster, at which time it is passed the `WhiteBoardClient` for which it will act as the service object. Creating the service object returns a `WhiteBoardPrivate` interface on the new service object.

```
    // Now introduce the new Whiteboard to its peers & masterlist

    synchronized(whiteboards){
      Enumeration e = whiteboards.elements();
      while(e.hasMoreElements()){
        WhiteBoardPrivate wb = (WhiteBoardPrivate)e.nextElement();
        wbp.addServiceObj(wb);
        wb.addServiceObj(wbp);
      }
      whiteboards.addElement(wbp);
    }
    return(wbp.getPublicInterface());
  }
}
```

The penultimate step is to add the new service object's `WhiteBoardPrivate` interface to all of its existing peers, and add the new object to the master list. The `WhiteBoardPrivate` interface is used within the application to control the service objects, and therefore must not be passed to any external entity. So the service object is called to obtain its `WhiteBoardShared` interface, which is returned to the client.

```
public void handOff(Contact m){}

public void resultNotification(LotReference l, Results r){
  synchronized(auctions){
    auctions.put(l,r);
    auctions.notifyAll();
  }
}
```

The `handOff` and `resultNotification` methods are required by all market clients implementing the `Vendor` interface. Only the `resultNotification` method is of interest in this application, and is used by the market to communicate the results of each auction to the application. The actions of this method are to, within the application contact object, update the `auctions` vector to store a new result, and then wake all waiting threads so that they may check if it is their auction which has been completed.

8.4.3 Précis

Both client-server and distributed versions of the whiteboard application were simple to implement within the Nomad environment. The total programming time for both applications was under three hours, and debugging minimised due to the underlying transparent remote invocation system.

The additional sophistication of the distributed implementation is supported by the Nomad infrastructure. The resulting application separates the role of the contact object from that of the service objects and uses the Nomad Marketplace to locate vHosts for the service objects near the client.

Much of the additional code was due to the need for thread synchronisation with auction results, and coordinating application components. Specifically, negotiating for resources and creating distributed objects is straightforward, ensuring that building a distributed application within Nomad is relatively painless.

Chapter 9

Conclusions

The objective underlying the work in this thesis is to support mobility as a first class paradigm in a scalable, open system. This objective requires that a set of architectural goals are met. These are restated here from chapter 1.

1. Support mobility as a first class paradigm for the future:
   (a) provide distribution transparencies,
   (b) but permit contextual awareness (selective transparencies),
   (c) allow customisable middleware architecture,
   (d) support adaptive applications, and
   (e) meet QoS specifications.
2. Provide an open system:
   (a) globally allocate computational and other resources,
   (b) support and take advantage of heterogeneity,
   (c) control the behaviour of each mobile application with respect to the local goals and policies of the administrative domain being visited,
   (d) protect applications from malicious hosts, and
   (e) encourage the deployment and building of infrastructure.
3. Ensure scalability:

(a) design scalable global infrastructure,

(b) minimise overheads to both the system and its applications, and

(c) automate management where possible.

The Nomad architecture achieves virtually all of these goals, and this chapter reviews these contributions, followed by detail on future research, and finishes with some final observations.

9.1 Review

The entire architecture of Nomad is designed with mobility as the basic paradigm **1**, and as such, much of the Nomad infrastructure is intended to be mobile itself and adapt itself to conditions prevailing within the system. Mobility in applications is supported by the architectures of all three Nomad layers, that is, the application, service and Depot layers.

Distribution transparencies (access and location) **1(a)** are provided by default to applications by the vHosts on which they are executing and by the automatic rebinding of out-of-date references by proxies and the location service. The other standard transparencies of migration, relocation, persistence and transaction transparencies are incompatible with the application directed (as opposed to system directed) distribution model in Nomad. These services are instead provided as very high level facilities by the vHosts which the applications can use as they desire.

Applications may opt-out of the default distribution transparencies themselves **1(b)** if they wish to take advantage of collocation of execution with clients or data, by specifying location information when selecting Depots on which to execute. An application may also customise its environment **1(c)**, by supplying its own implementations of transaction and communications protocols and customising its location tables to meet its own functional requirements.

Adaptive applications **1(d)** which alter their structure or function in response to the availability of resources are catered for by the Nomad Marketplace resource negotiation system, which allows them to specify their resource requirements and to find Depots capable of satisfying or at least providing an acceptable compromise. Likewise, these

resource requirements include QoS specifications **1(e)** that the contracting Depot will honour.

Open systems have no control on who participates and any application may be run without prior approval. With Nomad, the *openness* of the system is extended to the inclusion of arbitrary Depots as well. The Nomad Market is designed to facilitate optimal global allocations of resources to applications **2(a)**, with the use of resource description graphs which enable applications to specify their needs and to take advantage of the heterogeneity within the system **2(b)**. To prevent applications from maliciously (or otherwise) over-utilising or wasting Depot resources, Nomad uses an economic resource model with incentive pricing to control application behaviour **2(c)**. The expectation is that applications will offer services to clients, and will require computational resources supplied by Depots. The resource infrastructure provided by the Depots needs to be built and the services need to be created before clients may use them. To encourage the deployment and building of both these forms of infrastructure **2(e)**, providers of infrastructure (Depots) and services (applications) earn convertible currency from their respective clients.

The ability to scale is of vital importance for both current and future distributed systems. Scale has been one of the most consistently underestimated aspects of system design, and as computing systems become ubiquitous the numbers of entities can only increase. All of the Nomad global services designed in this thesis have been created with the prospect of scaling over very large systems **3(a)**. In addition, the more load that a component must bear, the harder it is to scale, and therefore computational and communication overheads have been minimised **3(b)**. Lastly, as systems grow larger, the cost of human management becomes an ever greater problem. The explicit use of policy throughout the system is intended to minimise human involvement and management overheads **3(c)**.

9.1.1 The Negotiation Service

The Nomad negotiation service is a distributed resource allocation system, which enables applications to negotiate and pay for execution resources. The mechanism is simple, allowing computationally limited clients to construct resource descriptions from building blocks, and to negotiate simply and efficiently.

Negotiation

This thesis analyses resource negotiation for non-cooperative entities and determines that where there is competition between multiple producers or between multiple consumers, auction protocols are most efficient – both in the optimality of allocation and computationally. Another advantage of auctions within an economic resource economy is that they automatically determine the *market price* of a resource without computational effort from the vendor. Of the auction protocols, the best candidate was determined to be the Vickrey protocol, which is a single bid protocol in which truthful bidding is the dominant strategy. Truthful bidding eliminates counterspeculation and the computational overhead that it inflicts on the entire system.

Detailed analysis shows that the protocol has limitations in some situations. However, if its limitations are addressed by careful design of the protocol mechanism, then the Vickrey auction is an ideal basis for automation. The allocations made by the Vickrey protocol are Pareto-optimal, stable and individually rational as truthful bidding ensures that the payoff is not less than not participating.

Resource Description Graphs

A resource description graph (RDG) is a single graph which describes an auction and associated compromises. Each sentence through the graph identifies a non-unique set of resources required to fulfil that task, alternate sentences define acceptable compromises. Multiple lots can be specified in a RDG and a bid over all the lots in an auction, an entirety bid, can capture a degree of bidder synergy.

The RDG provides a general representation of resources, which permits the vendor or bidder to limit the permissible allocation combinations, and reduce the combinatorial allocation problem (CAP). Nomad goes further and utilises the RDG and entirety bidding to replace the *NP*-complete CAP with a good approximation enabling multi-component auctions with the standard Vickrey auction protocol.

Marketplace

Resource negotiation and allocation is carried out within the Nomad Marketplace, which incorporates the Vickrey auction protocol. The mechanism design in the Nomad Marketplace addresses all of the limitations of the Vickrey auction protocol discussed in the negotiation analysis.

Negotiation starts with an application specifying its resource requirements in a RDG which is then used by the Marketplace to create an auction. Depots that can satisfy the requirements bid on the auctions, with their true valuation of the resources. The resulting contract is awarded to the lowest bidder, at the price of the second lowest bid.

Coordination within the Marketplace is via the distribution of catalogues, and categories provide a means of advertising for specific types of reources. Bipartite auctions are simulated within the multipartite auction protocol, in such a way that the Depot will bid its true value.

The combination of the Vickrey auction, the resource description graphs and the Market mechanism provides an ideal and novel solution to the problem of distributed global resource allocation.

9.1.2 The Location Service

This thesis presents a novel solution to the design of a distributed location service for large scale mobile object systems. This approach uses the application to optimise the distribution of the location tables, and limits the impact of the majority of updates to these very small infrequently mobile data structures. The remaining global workload involves only resolving the locations of the location tables themselves.

Applications may customise their location tables to meet their own functional requirements, determine where the tables are located, the degree of replication and consistency protocols.

9.1.3 The Prototype

The prototype is a functional implementation of the Nomad architecture and demonstrates its feasibility. Within the prototype, distributed applications can be created which

negotiate for resources via the Marketplace and migrate through the system executing on different Depots. The services that they implement can be bound from outside and transparently rebound as the application moves. Implementation of such mobile distributed applications is straightforward and the high level infrastructure makes the design of such applications easier and reduces programming errors related to the subtleties of distribution.

9.2 Future Work

Future work is required on the Nomad architecture – while applications can be written and deployed, basic research is required to price infrastructure, form policy, verify existing design decisions, and to implement some of the neglected parts of the prototype.

9.2.1 Experimental Verification

The location service is claimed to scale due to its adoption of location tables. This needs to be confirmed via simulation, as testing within the prototype is insufficient to measure the full impact of scale.

The approximation to the solution of the combinatorial allocation problem utilising the RDG needs to be assessed and compared against the GVA and iBundle.

9.2.2 Additional Implementation

The Nomad prototype still requires the implementation of: persistence and quiescence, QoS scheduling and contract valuation.

A payment infrastructure needs to be designed and incorporated into the existing architecture. Such systems have been implemented previously in Tacoma [50] and D'Agents [14], and the underlying accounting system can be achieved relatively cheaply as demonstrated by Nemesis [88].

On a more pragmatic note, the prototype needs to be ported to a more recent version of Java, and some non-Java interpreters for vHosts are needed to provide alternate language support.

9.2.3 Additional Research

The RDG needs further work to fully understand edge expressions and the resource domain. Another project is the construction of an efficient and effective RDG pattern matching algorithm.

More work is needed to analyse the interactions of the entities within Nomad, and to then form the basis of an incentive pricing scheme.

The final problem in this section is a direct result of the inclusion of arbitrary Depots within the system. The problem is one of trust. That is, how can you trust an arbitrary Depot not to:

- steal data or funds from the applications it is hosting (protection), or
- not supply the resources it has contracted to provide (enforcement), or
- interfere with the information in the global structures which it may be hosting (corruption).

This list is not exhaustive, it does however highlight some of the problems inherent in allowing the 'openness' of the system to extend to Depots. Potential solutions include obscurification of application objects or to make 'trust' a negotiable quantity. One Depot might be run by a reputable company (TrustMeCo), and run a signed implementation of the Depot with a manual certification that it is trustworthy. It would be better to trust your company's sensitive data to Depots run by TrustMeCo, rather than an arbitrary unknown, and this would be included in the resource negotiation.

Provision of this trust status would also require certification from a trusted third party, as untrustworthy machines cannot be trusted to correctly reveal their own status. Trustworthy Depots could expect to charge a premium for their resources.

9.3 Summary

This thesis draws together disparate areas of research in a novel approach to distributed system architecture, and documents the resulting design and development of the system along with the implementation of a prototype. The architecture captures all of the salient

points of mobile system design, and contributes significantly to the understanding of how to design and implement large scale computer systems.

The Nomad service layer, incorporating the location and negotiation services, provides a unifying view of the system for both applications and Depots. With the combination of this and the Depot and application architectures, Nomad represents a middleware architecture that meets the majority of the current architectural challenges for the future.

Nomad belongs to a group of essential, but as yet unavailable distributed system architectures. Until such systems become available, application designs will remain limited in scope, and unable to take advantage of new programming structures and paradigms such as, mobile agents, distributed objects or adaptive applications.

Appendix A

Interfaces

A.1 Location Service

```
public interface LocationTable{
  public MobileClusterName resolve(GlobalID clusterID) throws NotFoundException
  public boolean map(GlobalID clusterID, MobileClusterName newName, Authority auth);
  public void unmap(GlobalID clusterID, Authority auth);
  public void move(VHost dest, Authority auth) throws MoveFailedException;
}
```

A.2 Market

A.2.1 Clients

```
public interface Client{
  public void handOff(Contact m);  // use this MarketPlace instead.
  public void resultNotification(LotReference l, Results r);
}
```

Application (Vendor)

```
public interface Vendor extends Client{
  // Nothing additional at present.
}
```

Depot (Bidder)

```
public interface Bidder extends Client{
  public void requestBid(Catalog c); // MarketPlace asks a bidder to bid
                                     // on a particular lot.  This catalog
                                     // only contains the lot in question
}
```

A.2.2 Market

```
public interface Contact{
  public Bidding amBidder(Bidder b);
  public Bidding amBidder(Bidder b, ResourceDescription rd);
  public Vending amVendor(Vendor v);
}


public interface Participant{
  public AuctionStatus queryAuction(AuctionReference l);
}


public interface Vending extends Participant{
  public AuctionReference placeAuction(Vendor v, String rd, AuctionProperties p);
  public AuctionStatus withdrawAuction(AuctionReference l);
}


public interface Bidding extends Participant{
  public Catalog getCatalog();  // returns JIT window Auctions
  public Catalog getCatalog(ResourceDescription rd); // returns pattern matches

  public BidReference placeBid(Bidder b, AuctionReference l, Bid bid);
  public BidStatus withdrawBid(BidReference cRef);
}
```

Appendix B

Whiteboard Implementation

B.1 Whiteboard Client Implementation

```
package nz.ac.vuw.nomad.applications.whiteboard;

// Same for Client-Server and Distributed implementations
// uses a thread every 5s to send updates to the WB.

import nz.ac.vuw.nomad.mobility.AuthMobileObject;
import nz.ac.vuw.nomad.mobility.VHost;
import nz.ac.vuw.nomad.services.IfaceRepository;
import java.lang.Thread;
import java.lang.String;

public class WhiteBoardClientImp extends AuthMobileObject
  implements WhiteBoardClient{

  public WhiteBoardShared wbs = null;
  public String name = "";

  // AuthMobileObject methods

  public WhiteBoardClient init(String name){
    // Now get an interface on a whiteboard from the YellowPages

    try{
    IfaceRepository lir = ((VHost)getPlace()).getLocalInterfaceRepository();
    IfaceRepository yp = (IfaceRepository)lir.get("/nomad/services/yellowPages");
    ContactObject wb = (ContactObject)yp.get("/nomad/applications/whiteboard");

    this.wbs = wb.getWhiteBoard((WhiteBoardClient)this);
    this.name = name;

    // Now wbs is an interface on the shared whiteboard.  Use a thread
    // to write test messages.

    ClientThread t = new ClientThread(wbs, name);
    t.start();
```

```
    }catch(Exception e){
      e.printStackTrace();
    }

    return(WhiteBoardClient)this;
  }

  public void restart(Exception e){
    // Seem to have moved.  Restart test thread.

    ClientThread t = new ClientThread(wbs, name);
    t.start();
  }

  // WhiteBoardClient method

  public void write(String s){
    // updates from the shared whiteboard, update local state.

    System.out.println("WB: "+s);
  }
}

class ClientThread extends Thread{
  long period = 5*1000;
  int count = 0;
  WhiteBoardShared client = null;
  String name = "";

  public ClientThread(WhiteBoardShared client, String name){
    this.client=client;
    this.name=name;
  }

  public void run(){
    // send test messages to the shared whiteboard every 5 seconds.

    while(true){
      try{
        sleep(period);
        client.write(name + ": call("+ String.valueOf(count++) +")");
      }catch(InterruptedException e){
        client.write(name + ": call("+ String.valueOf(count++) +")");
      }
    }
  }
}
```

B.2 Whiteboard Service Implementation

```
package nz.ac.vuw.nomad.applications.whiteboard;

import nz.ac.vuw.nomad.mobility.AuthMobileObject;
import java.lang.String;
import java.util.Vector;
```

```
import java.util.Enumeration;

public class WhiteBoardService extends AuthMobileObject
  implements WhiteBoardPrivate, WhiteBoardShared{

  public WhiteBoardClient wbc = null;
  public Vector peers = null;

  // AuthMobileObject methods

  public WhiteBoardPrivate init(WhiteBoardClient wbc){
    this.wbc = wbc;
    peers = new Vector();
    return(WhiteBoardPrivate)this;
  }

  public void restart(Exception e){}

  // WhiteBoardShared methods.

  public void write(String s){
    // Forward updates from client to all peers and client.

    wbc.write(s);
    Enumeration e = peers.elements();
    while(e.hasMoreElements())
      ((WhiteBoardPrivate)e.nextElement()).peerWrite(s);
  }

  // WhiteBoardPrivate methods

  public void serviceWrite(String s){
    // update from peer (shared whiteboard) write to client.

    wbc.write(s);
  }
  public void addServiceObj(WhiteBoardPrivate wbp){
    // add a new peer to our peer list.

    peers.addElement(wbp);
  }

  public WhiteBoardShared getPublicInterface(){
    return(WhiteBoardShared)this;
  }
}
```

References

[1] Paulo Amaral, Christian Jacquemot, Peter Jensen, Rodger Lea, and Adam Mirowski. Transparent Object Migration in COOL-2. Technical Report CS/TR-92-30, Chorus Systèmes, 1992.

[2] Yeshayahu Artsy and Raphael Finkel. Designing a Process Migration Facility, The Charlotte Experience. *IEEE Computer Magazine*, 22(9):47–56, September 1989.

[3] A. Baggio, G. Ballintijn, M. van Steen, and A.S. Tanenbaum. Efficient Tracking of Mobile Objects in Globe. *The Computer Journal*, 44(5):340–353, 2001.

[4] G. Ballintijn, M. van Steen, and A.S. Tanenbaum. Scalable User-Friendly Resource Names. *IEEE Internet Computing*, 5(5):20–27, 2001.

[5] Gerco Ballintijn, Maarten van Steen, and Andrew S. Tanenbaum. Exploiting Location Awareness for Scalable Location-Independent Object IDs. In *Proceedings of the Fifth Annual ASCI Conference*, pages 321–328, Heijen, The Netherlands, June 1999. Delft University of Technology.

[6] Amnon Barak, Avner Braverman, Ilia Gilderman, and Oren Laadan. Performance of PVM with the MOSIX Preemptive Process Migration. In *Proceedings of the 7th Israeli Conference on Computer Systems and Software Engineering*, pages 38–45, Herzliya, 1996.

[7] Amnon Barak and Ami Litman. MOS: A Multi-computer Distributed Operating System. *Software—Practice and Experience*, 15(8):724–737, August 1985.

[8] Massimo Bartoli, Marcello Pelillo, Kaleem Siddiqi, and Steven W. Zucker. Attributed Tree Homomorphism Using Association Graphs. In *Proceedings of the 15th International Conference on Pattern Recognition*, volume 2, pages 133–136,

Barcelona, Spain, September 2000. International Association for Pattern Recognition.

[9] T. Berners-Lee, L. Masinter, and M. McCahill. Uniform Resource Locators (URL). Network Working Group, Request for Comments (RFC) 1738, December 1994.

[10] Brian Bershad. Load Balancing with Maitre d'. Technical Report CSD-85-276, University of California at Berkeley, December 1985.

[11] Azer Bestavros. Speculative Data Dissemination and Service. In Stanley Y. W. Su, editor, *Proceedings of the Twelfth International Conference on Data Engineering (ICDE)*, pages 180–187, New Orleans, Louisiana, February 26 - March 1 1996. IEEE Computer Society.

[12] David L. Black. Scheduling Support for Concurrency and Parallelism in the Mach Operating System. *IEEE Computer Magazine*, 23(5):35–43, May 1990.

[13] Andy Bond and John H. Hine. DRUMS: A Distributed Performance Information Service. In J. Lions, editor, *Proceedings of the 14th Australian Computer Science Conference*, Sydney, February 1991.

[14] Jonathan Bredin, David Kotz, and Daniela Rus. Market-Based Resource Control for Mobile Agents. In *Proceedings of the Second International Conference on Autonomous Agents*, pages 197–204. ACM Press, May 1998.

[15] Jonathan Bredin, David Kotz, and Daniela Rus. Economic Markets as a Means of Open Mobile-Agent Systems. In *Proceedings of the Workshop Mobile Agents in the Context of Competition and Cooperation (MAC3) at Autonomous Agents 99*, pages 43–49, Seattle, Washington, May 1999.

[16] Jonathan L. Bredin. *Market-based Control of Mobile-agent Systems.* PhD thesis, Department of Computer Science, Dartmouth College, June 2001. Available as Dartmouth Computer Science Technical Report TR2001-408.

[17] Kris Bubendorfer and John H. Hine. DepotNet: Support for Distributed Applications. In *Proceedings of INET'99, Internet Society's 9th Annual Networking Conference*, June 1999.

[18] Kristian P. Bubendorfer. Resource based policies for load distribution. Master's thesis, Victoria University of Wellington, August 1996.

[19] Roderic Geoffrey Galton Cattell. *Object Data Management: object-oriented and extended relational database systems.* Addison-Wesley, 1991.

[20] Dan Chalmers and Morris Sloman. Survey of Quality of Service in Mobile Computing Environments. Technical Report 98/10, Department of Computing, Imperial College, London, February 1999.

[21] Steve Chapin, Dimitrios Katramatos, John Karpovich, and Andrew Grimshaw. Resource Management in Legion. Technical Report CS-98-09, Department of Computer Science, University of Virgina, 1998.

[22] D. R. Cheriton. The V Distributed System. *Communications of the ACM*, 31(3):314–333, March 1988.

[23] Douglas E. Comer. *Computer Networks and Internets.* Prentice-Hall, 1st edition, 1997.

[24] Ezra E. K. Cooper and Robert S. Gray. An Economic CPU-Time Market for D'Agents. Technical Report TR2000-375, Dartmouth College, Computer Science, Hanover, NH, June 2000.

[25] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design.* Pearson Education Ltd, 3rd edition, 2001.

[26] John Creedy, Jeff Borland, and Jürgen Eichberger, editors. *Recent Developments in Game Theory.* Edward Elgar Publishing, 1992.

[27] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. PONDER: A Language for Specifying Security and Management Policies for Distributed Systems. Technical Report DoC 2000/1, Imperial College of Science Technology and Medicine, 2000.

[28] Sven de Vries and Rakesh V. Vhora. Combinatorial Auctions: A Survey. *INFORMS Journal of Computing*, to appear.

[29] Frederick Douglis. *Transparent Process Migration in the Sprite Operating System.* PhD thesis, Graduate Division of the University of California at Berkeley, 1990.

[30] Jon Doyle. A Reasoning Economy for Planning and Replanning. In *Technical Papers of the ARPA Planning Initiative Workshop*, Tucson, Arizona, February 1994.

[31] K. Eric Drexler and Mark S. Miller. Incentive Engineering for Computational Resource Management. In Huberman B.A, editor, *The Ecology of Computation*, pages 231–267. Elsevier Science Publishers (North-Holland), 1988.

[32] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: an Operating System Architecture for Application-level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 251–266, Copper Mountain Resort, Colorado, December 1995.

[33] Kurt Geihs. Middleware Challenges Ahead. *IEEE Computer Magazine*, 34(6):24–31, June 2001.

[34] Pawan Goyal, Xingang Guo, and Harrick M. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *Proceedings of the USENIX 2nd Symposium on Operating System Design and Implementation*, Seattle, Washington, October 1996.

[35] Andrew Grimshaw, Adam Ferrari, Frederick Knabe, and Marty Humphrey. Legion: An Operating System for Wide Area Computing. Technical Report CS-99-12, Department of Computer Science, University of Virgina, 1999.

[36] Andrew Grimshaw, Michael Lewis, Adam Ferrari, and John Karpovich. Architectural Support for Extensibility and Autonomy in Wide-Are Distributed Object Systems. Technical Report CS-98-12, Department of Computer Science, University of Virgina, 1998.

[37] Mor Harchol-Balter and Allen B. Downey. Exploiting Process Lifetime Distributions for Dynamic Load Balancing. *ACM Transactions on Computer Systems*, 15(3):253–285, 1997.

[38] Garrett Hardin. The Tragedy of the Commons. *Science*, 162:1243–1248, 1968.

[39] F.J. Hauck, M. van Steen, and A.S. Tanenbaum. A Location Service for Worldwide Distributed Objects. In *Proceedings of the 10th European Conference on Object-Oriented Programming (ECOOP)*, Linz, Austria, July 1996.

[40] Richard Hayton and the Advanced Networked Systems Architecture Team. *FlexiNet Architecture.* Citrix Systems (Cambridge) Limited, Poseidon House, Castle Park, Cambridge, CB3 0RD, United Kingdom, February 1999.

[41] P. Homburg, L. van Doorn, M van Steen, A. S. Tanenbaum, and W. de Jonge. An Object Model for Flexible Distributed Systems. In *Proceedings of the First Annual ASCI Conference*, Heijen, Netherlands, May 1995. Delft University of Technology.

[42] P. Homburg, M. van Steen, and A.S. Tanenbaum. An Architecture for a Wide Area Distributed System. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, Connemara, Ireland, September 1996. ACM SIGOPS.

[43] P. Homburg, M. van Steen, and A.S. Tanenbaum. Communication in GLOBE: An Object-Based Worldwide Operating Systems. In *Proceedings of the Fifth International Workshop on Object Orientation in Operating Systems*, Seattle, Washington, October 1996.

[44] P. Homburg, M. van Steen, and A.S. Tanenbaum. Unifying Internet Services Using Distributed Shared Objects. Technical Report IR-409, Department of Computer Science, Vrije Universiteit, Amsterdam, October 1996.

[45] Leonid Hurwicz, David Schmeidler, and Hugo Sonnenschein, editors. *Social Goals and Social Organization: Essays in Memory of Elisha Pazner*, chapter The Economics of Competitive Bidding: A Selective Survey, Paul R. Milgrom, pages 261–289. Cambridge University Press, 1985.

[46] Norman C. Hutchinson. *Emerald: An Object Based Language for Distributed Programming.* PhD thesis, Department of Computer Science, University of Washington, 1987.

[47] Renato Iannella, Hoylen Sue, and Danny Leong. BURNS: Basic URN Service Resolution for the Internet. In *Proceedings of the Asia-Pacific World Wide Web Conference*, Beijing and Hong Kong, August 1996.

[48] David Ingram. *Integrated Quality of Service Management.* PhD thesis, Jesus College, University of Cambridge, January 2000.

[49] International Standards Organisation. *Open Distributed Processing Reference Model — Part 3: Architecture, ISO/IEC IS 10746-3*, 1995.

[50] Dag Johansen, Robbert van Renesse, and Fred B. Schneider. Operating System Support for Mobile Agents. In *Proceedings of the 5th IEEE Workshop on Hot Topics in Operating Systems*, 1997.

[51] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[52] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application Performance and Flexibility on Exokernel Systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 52–65, Saint-Malô, France, October 1997.

[53] Eleni Kaldoudi, Marios Zikos, E. Leisch, and Stelios C. Orphanoudakis. PLEGMA: An Agent-Based Architecture for Developing Network-Centric Information Processing Services. Technical report, Institute of Computer Science, Foundation for Research and Technology - Hellas, September 1998.

[54] Gregory E. Kersten and Sunil J. Noronha. Supporting International Negotiations with a WWW-based System. Technical Report INR05/97, International Institute for Applied Analysis, Laxenburg, Austria, 1997.

[55] Eric Korpela, Dan Werthimer, David Anderson, Jeff Cobb, and Matt Lebofsky. SETI@home: Massively Distributed Computing for SETI. *IEEE: Computing in Science and Engineering*, 3(1):78–83, 2001.

[56] David Kotz and Robert S. Gray. Mobile Agents and the Future of the Internet. *ACM Operating Systems Review*, 33(3):7–13, August 1999.

[57] Ryszard Kowalczyk and Van Bui. On Fuzzy E-Negotiation Agents: Autonomous Negotiation with Incomplete and Imprecise Information. In *Proceedings of the DEXA e-Negotiation Workshop*, 2000.

[58] Ben J. A. Kröse and P. Patrick van der Smagt. *An Introduction to Neural Networks.* The University of Amsterdam, 1993.

[59] James F. Kurose and Rahul Simha. A Microeconomic Approach to Optimal Resource Allocation in Distributed Computer Systems. *IEEE Transactions on Computers*, 38(5):705–717, May 1989.

[60] Danney B. Lange. Java Aglet Application Programming Interface. Technical Report J-AAPI, IBM Tokyo Research Laboratory, 1997.

[61] Rodger Lea, Christian Jacquemot, and Eric Pillenvesse. COOL: System Support for Distributed Object-Oriented Programming. *Communications of the ACM*, 36(9), sept 1993.

[62] Henry M. Levy and Ewan D. Tempero. Modules, Objects and Distributed Programming: Issues in RPC and Remote Object Invocation. *Software Practice and Experience*, 21(1):77–90, January 1991.

[63] Micheal J. Litzkow, Miron Livny, and Matt W. Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th IEEE International Conference on Distributed Computing Systems*, pages 104–111, Los Alamitos, California, 1988. IEEE, IEEE CS Press.

[64] Tyng-Luh Liu and David Geiger. Approximate Tree Matching and Shape Similarity. In *Proceedings of the 7th IEEE International Conference on Computer Vision*, Kerkyra, Greece, September 1999. IEEE.

[65] Jeffery K. MacKie-Mason and Hal R. Varian. Generalized Vickrey Auctions. Working paper, University of Michigan, 1994.

[66] Thomas W. Malone, Richard E. Fikes, Kenneth R. Grant, and Michael T. Howard. Enterprise: A Market-like Task Scheduler for Distributed Computing Environments. In Huberman B.A, editor, *The Ecology of Computation*, pages 177–205. Elsevier Science Publishers (North-Holland), 1988.

[67] Paul Milgrom and Robert Weber. A Theory of Auctions and Competitive Bidding. *Econometrica*, 50(5):1089–1122, September 1982.

[68] Barton P. Miller, David L. Presotto, and Michael L. Powell. DEMOS/MP: The Development of a Distributed Operating System. *Software Practice and Experience*, 17(4):277–290, April 1987.

[69] Mark S. Miller and K. Eric Drexler. Markets and Computation: Agoric Open Systems. In Huberman B.A, editor, *The Ecology of Computation*, pages 133–176. Elsevier Science Publishers (North-Holland), 1988.

[70] R. Moats. URN Syntax. Network Working Group, Request for Comments (RFC) 2141, May 1997.

[71] P. Mockapetris. Domain Names: Concepts and Facilities. Network Working Group, Request for Comments (RFC) 1034, November 1987.

[72] Peter Morris. *Introduction to Game Theory.* Springer, 1994.

[73] Michael N. Nelson, Graham Hamilton, and Yousef A. Khalidi. A Framework for Caching in an Object-Oriented System. Technical Report SMLI TR-93-19, Sun Microsystems Laboratories, 1995.

[74] Tsuyoshi Ohta, Takashi Watanabe, and Tadanori Mizuno. A Job Dependent Dispatching Scheme in a Heterogeneous Multiserver Network. *IEICE Transactions on Communications*, E77-B(11):1380–1387, November 1994.

[75] John K. Ousterhout, Donald A. Scelza, and Pradeep S. Sindu. Medusa: An Experiment in Distributed Operating System Structure. *Communications of the ACM*, 23(2):92–105, February 1980.

[76] David C. Parkes. iBundle: An efficient ascending price bundle auction. In *Proceedings of the 1st ACM Conference on Electronic Commerce, EC-99*, pages 148–157, 1999.

[77] David C. Parkes. An Iterative Generalized Vickrey Auction: Strategy-Proofness without Complete Revelation. In *Proceedings of the AAAI Spring Symposium on Game Theoretic and Decision Theoretic Agents*, Stanford University, CA, March 2001.

[78] David C. Parkes, Jayant Kalagnanam, and Marta Eso. Achieving Budget-Balance with Vickrey-Based Payment Schemes in Exchanges. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1161–1168, 2001.

[79] Marcello Pelillo, Kaleem Siddiqi, and Steven W. Zucker. Attributed Tree Matching and Maximum Weight Cliques. In *Proceedings of ICIAP'99 — the 10th International Conference on Image Analysis and Processing*, Venice, Italy, September 1999.

[80] Marcello Pelillo, Kaleem Siddiqi, and Steven W. Zucker. Continuous-based Heuristics for Graph and Tree Isomorphisms, with Application to Computer Vision. In *Proceedings of the Conference on Approximation and Complexity in Numerical Optimization: Continuous and Discrete Problems*, University of Florida, February 1999.

[81] Marcello Pelillo, Kaleem Siddiqi, and Steven W. Zucker. Matching Hierarchical Structures Using Association Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 21(11):1105–1120, 1999.

[82] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel. LOCUS: A Network Transparent, High Reliability Distributed System. In *Proceedings of the 8th Symposium on Operating System Principles*, pages 169–177, Pacific Grove, CA, December 1981. ACM.

[83] M. L. Powell and B. P. Miller. Process Migration in DEMOS/MP. In *Proceedings of the 9th Symposium on Operating Systems. In ACM Operating Systems Review 17(5)*, pages 110–119, 1983.

[84] Ragunathan Rajkumar, Chen Lee, John Lehoczky, and Dan Siewiorek. A Resource Allocation Model for QoS Management. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, San Francisco, California, December 1997.

[85] A. Rangarajan and E. Mjolsness. A Lagrangian Relaxation Network for Graph Matching. *IEEE Transactions on Neural Networks*, 7(6):1365–1381, 1996.

[86] Richard F. Rashid. Experiences with the Accent Operating System. In Günter Müller and Robert P. Blanc, editor, *Proceedings of Networking in Open Systems: International Seminar*, pages 252–269, Oberlech, Austria, August 1986. IBM European Networking Center (Heidelberg, Germany) and United States. National Bureau of Standards., Springer-Verlag, Berlin Heidelberg.

[87] E. Rasmusen. *Games and Information — An Introduction to Game Theory*. Blackwell Publishers, Oxford, 2nd edition, 1994.

[88] Dickon Reed, Ian Pratt, Paul Menage, Stephen Early, and Neil Stratford. Xenoservers: Accounted Execution of Untrusted Code. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HotOS-VII)*, Rio Rico Resort and Country Club, Rio Rico, Arizona, March 1999. IEEE Computer Society and IEEE Technical Committee on Operating Systems (TCOS).

[89] M. H. Rothkopf, A. Pekeč, and R. M. Harstad. Computationally Manageable Combinatorial Auctions. Technical Report 95-09, DIMACS, Center for Discrete Mathematics and Theoretical Computer Science, Rutgers, New Jersey, USA, April 1995.

[90] Tuomas W. Sandholm. An Implementation of the Contract Net Protocol Based on Marginal Cost Calculations. In *Proceedings of the 12th International Workshop on Distributed Artificial Intelligence*, pages 295–308, Hidden Valley, Pennsylvania, 1993.

[91] Tuomas W. Sandholm. Issues in Automated Negotiation and Electronic Commerce: Extending the Contract Net Framework. In *Proceedings of the First International Conference on Multiagent Systems (ICMAS-95), San Francisco, California*, pages 328–335, Menlo park, California, June 1995. AAAI Press / MIT Press.

[92] Tuomas W. Sandholm. Limitations of the Vickrey Auction in Computational Multiagent Systems . In *Proceedings of the Second International Conference on Multiagent Systems (ICMAS-96)*, pages 299–306, Keihanna Plaza, Kyoto, Japan, December 1996.

[93] Tuomas W. Sandholm. *Negotiation Among Self-Interested Computationally Limited Agents*. PhD thesis, Department of Computer Science, University of Massachusetts, September 1996.

[94] Richard E. Schantz, Robert H. Thomas, and Girome Bono. The Architecture of the Cronus Distributed Operating System. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, pages 250–259, Cambridge, MA., USA, May 1986. IEEE.

[95] Edward C. Slottow. Engineering a Global Resolution Service. Master's thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, June 1997.

[96] Jonathan M. Smith. A Survey of Process Migration Mechanisms. *Operating Systems Review*, 22(3):28–40, July 1988.

[97] Reid G. Smith. The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver. *IEEE Transactions on Computers*, 29(12):1104–1113, December 1980.

[98] K. Sollins and L. Masinter. Functional Recommendations for Internet Resource Names. Network Working Group, Request for Comments (RFC) 1737, December 1994.

[99] Karen. R. Sollins. Requirements and a Framework for URN Resolution Systems. Internet Engineering Task Force (IETF) Internet-Draft, March 1997.

[100] Neil Stratford and Richard Mortier. An Economic Approach to Adaptive Resource Management. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems*, Rio Rico Resort and Country Club, Rio Rico, Arizona, March 1999. IEEE Computer Society and IEEE Technical Committee on Operating Systems (TCOS).

[101] L. E. Sutherland. A Futures Market in Computer Time. *Communications of the ACM*, 11(6):449–451, June 1968.

[102] Andrew S. Tanenbaum. *Computer Networks*. Prentice-Hall, 3rd edition, 1996.

[103] Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, Sape J. Mullender, Jack Jansen, and Guido van Rossum. Experiences with the Amoeba Distributed Operating System. *Communications of the ACM*, 33:46–63, December 1990.

[104] Maarten van Steen, Franz J. Hauck, Philip Homburg, and Andrew S. Tanenbaum. Locating Objects in Wide-Area Systems. *IEEE Communications Magazine*, 36(1):104–109, January 1998.

[105] Maarten van Steen, Philip Homburg, and Andrew S. Tanenbaum. The Architectural Design of Globe: A Wide-Area Distributed System. Technical Report IR-422, Department of Computer Science, Vrije Universiteit, 1997.

[106] Maarten van Steen, Philip Homburg, and Andrew S. Tanenbaum. Globe: A Wide-Area Distributed System. *IEEE Concurrency*, 7(1):70–78, January–March 1999.

[107] William Vickrey. Counterspeculation, Auctions, and Competitive Sealed Tenders. *The Journal of Finance*, 16(1):8–37, March 1961.

[108] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A Note on Distributed Computing. Technical Report SMLI TR-94-29, Sun Microsystems Laboratories, 1995.

[109] Carl A. Waldspurger, Tad Hogg, Bernardo A. Huberman, Jeffrey O. Kephart, and W. Scott Stornetta. Spawn: A Distributed Computational Economy. *IEEE Transactions on Software Engineering*, 18(2):103–117, 1992.

[110] Michael P. Wellman. A Market-Oriented Programming Environment and its Application to Distributed Multicommodity Flow Problems. *Journal of Artificial Intelligence Research*, 1(1):1–23, 1993.

[111] Michael P. Wellman, William E. Walsh, Peter R. Wurman, and Jeffrey K. MacKie-Mason. Auction Protocols for Decentralized Scheduling. *Games and Economic Behavior*, 35(1/2):271–303, 2001.

[112] Jianxin Yan, Stephen Early, and Ross Anderson. The XenoService – A Distributed Defeat for Distributed Denial of Service. Technical report, Computing Laboratory, Cambridge, UK, 2000.