NOMAD: Application Participation in a Global Location Service

Kris Bubendorfer and John H. Hine

School of Mathematical and Computing Sciences, Victoria University of Wellington, P.O.Box 600, Wellington 6001, New Zealand
Kris.Bubendorfer@vuw.ac.nz, John.Hine@vuw.ac.nz

Abstract. The rapid growth of the world wide web has led to the wider appreciation of the potential for Internet-based global services and applications. Currently services are provided in an *ad hoc* fashion in fixed locations, which users locate and access manually. Mobility adds a new dimension to the location of such services in a global environment. Most systems supporting mobility rely on some form of home-based redirection which results in unacceptable residual dependencies. Nomad is a middleware platform for highly mobile applications. A significant contribution of the Nomad platform is a novel global object location service that involves the participation of both applications and global structures.

1 Introduction

Mobile code is a powerful paradigm for the creation of distributed software [1]. Applications constructed of mobile code are able to execute more efficiently, exploit heterogeneity and dynamically adapt to changes in their environment. The NOMAD (Negotiated Object Mobility, Access and Deployment) [2] architecture is a platform that provides a distributed systems infrastructure to support applications constructed of mobile code. The challenges faced by such applications include: the discovery of resources required for execution, the establishment of rights to access these resources, and the ability of an application to locate and coordinate its component parts. These issues are addressed by three complementary platform elements. Firstly NOMAD embodies an electronic Marketplace, through which applications locate and obtain the resources they require. Secondly, applications form contracts with the virtual machines on which they execute, specifying the quality of service to be provided. Lastly, NOMAD provides a global location service, to track the migration of code throughout the system, and to enable the various components of an application to coordinate to satisfy their collective goals. The ability to rapidly and reliably locate a mobile object or other resource is a critical and limiting factor in the growth of any large scale distributed system.

Location service research has previously concentrated on architectures with a geographical or organisational basis [3–7]. Most middleware location services, such as those offered by CORBA and Java RMI, do not address the question of mobility on a global scale, nor solve problems with residual dependencies. Other technologies such as Mobile IP [8] do not address location independence, transparency or residual dependencies.

This paper presents the Nomad location service, the design of which involves the participation of both applications and global infrastructure.

2 An Overview of the Nomad Architecture

NOMAD consists of loosely coupled cooperating virtual machines, named Depots, that host distributed applications. Each Depot independently provides a collection of local resources ranging from CPU cycles through to consistency protocols, which are made available to applications in return for payment. Applications utilise these resources to provide services to their clients, and negotiate to alter their resource profile as the demand for their services changes.

An application within NOMAD is a logical grouping of mobile code that performs one or more services (functions) for some client (another application, user etc.), employing the resources of one or more Depots. An application can range from a single mobile agent, implemented within a single piece of code and offering a single service, through to a collection of many pieces of mobile code offering many different services. The distinction is that an *application* is an autonomous unit of administration and identity, that is, the parts are cohesive and managed collectively.

Many language based objects, such as integers and strings, are too small to be independently mobile. An application's objects are arranged (by the programmer) into Clusters. The Cluster is a sophisticated wrapper that provides both mechanisms for mobility, and isolation of namespaces. References within a Cluster are handled by standard run-time support for intraprocess communication, while all references outside the Cluster take place as interprocess communications via proxies.

An application is responsible for: initiating negotiation for resources, directing the distribution of its *own* clusters, handling unrecoverable failures, and maintaining part of the location service. Each application controls its degree of distribution by placing and replicating its clusters to achieve QoS goals such as reduced latency to clients, redundancy and concurrent processing. The application negotiates with Depots via the NOMAD Marketplace for the resources that it requires. For their part, Depots use policies [9] to price their resources and to organise such things as the balance of work amongst a federation of Depots, or the quality of service ranges that individual Depots offer. A federation of Depots (as shown in Fig. 1) reflects common ownership, or administration of the Depots.

An application within Nomad consists of at least: a Contact Object (CO), Service Object (SO) and Location Table (LT). Figure 1 illustrates a client (C1) bound to the CO of Application 1, with which it must negotiate in order to obtain a service object. This situation is shown by client (C2), which is bound to the SO of Application 2 and from which it is receiving the requested service.

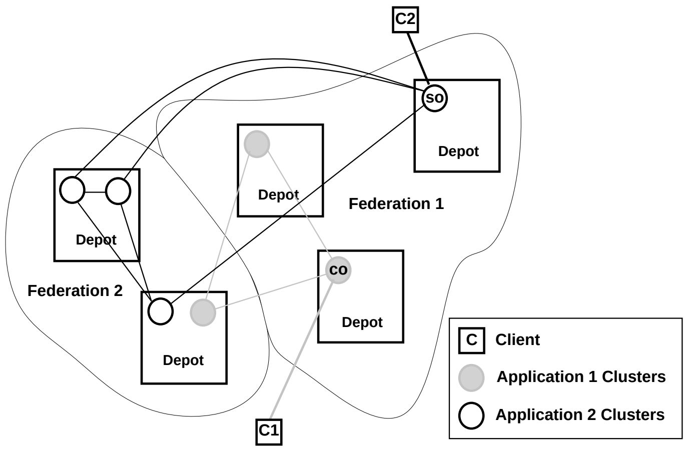

Fig. 1. *Interactions.*

Each Depot [10] consists of a Depot Manager and a set of managed virtual hosts (vHosts). Management policies dictate how each Depot will react to specific circumstances: how it will ensure levels of service and how it interacts with Depots within and outside of its federation. The Depot Manager is also responsible for responding to negotiation, charging, and arranging the hosting of the global NOMAD infrastructure components. A vHost is a virtual machine on which application code executes. A vHost is responsible for managing physical resource use, security, fault detection, and communication.

The Marketplace provides a set of services allowing applications to discover available resources and to contract with Depots for the provision of those resources. Negotiation within the Marketplace utilises the the Vickrey auction [11] protocol. Applications requiring a contract construct a description of their requirements in a resource description graph.

The combination of the consistent local infrastructure provided by the Depot architecture, and the global infrastructure provided by NOMAD services, combine to form a consistent and interlocking environment for distributed mobile applications. The location service provides dynamic binding to mobile objects and the initial discovery of services — by providing a reference for an appropriate contact object.

The remainder of this paper includes section 3.1, a short introduction on naming that illustrates how the Nomad location service, integrates with the proposed Internet Engineering Task Force uniform resource naming scheme (URN). The contribution of this paper, however, is in the design of the Nomad location service, and this starts with the underlying motivation in section 3.2.

3 The Nomad Location Service

A location service needs to track the location of each mobile object throughout that object's lifetime. In a large scale distributed system the location service may have to deal with billions 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. The general capabilities required for 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, an application should be able to locate and bind to all of its element objects, and the service should scale.

3.1 Naming

The URN [12–14] 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 [12].*

URNs are intended to make it easy to map other namespaces (legacy and future support) into the 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 [15].

Each URN has the structure `<URN> ::= "urn:"<NID>":"<NSS>`. The `NID` (namespace identifier) distinguishes between different namespaces, or contexts, which may be administered by different authorities. Such authorities could include ISO, ISBN, etc. The `NSS` (namespace specific string) is only valid within its particular `NID` context, and has no predefined structure. As pointed out in [13], 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.

URN resolution can be broken down into two steps; finding a resolver for a particular URN, and then resolving that URN. Finding a specific resolver requires a Resolver Discovery Service (RDS) [16, 14] as shown in Fig. 2. It is important to emphasise that the RDS need not cope with the high degrees of mobility, as found in Nomad, but rather must cope with the lesser mobility of NID resolvers.

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 paper 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.

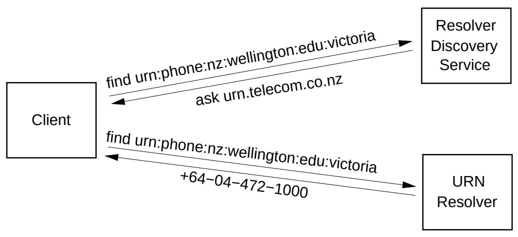
Client
Resolver Discovery Service
URN Resolver
find urn:phone:nz:wellington:edu:victoria
ask urn.telecom.co.nz
find urn:phone:nz:wellington:edu:victoria
+64−04−472−1000

Fig. 2. *Resolver Discover Service.*

3.2 Exploiting Application Locality

Existing systems, such as those cited in Sect. 1, 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. Using an organisational basis is also problematic for the same reasons.

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 a related 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 on behalf of itself, the Nomad system and its clients. In order to do this, the application maintains a set of location tables (LTs) that hold the current locations of all of the application's Clusters, and consequently their objects. In addition to providing locality of reference, an application's 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.

3.3 Design Overview

The Nomad location service must meet two distinct requirements; the discovery of services and the resolution of out-of-date bindings. Discovery provides an external interface used to establish an initial binding, while rebinding occurs internally and transparently. These two systems act independently but synergistically and are illustrated in Fig. 3. In addition, this figure highlights the separation between the Nomad level services above the dividing line, and the application level ones below.

The following list is a brief overview of the various location service components which feature in Fig. 3:

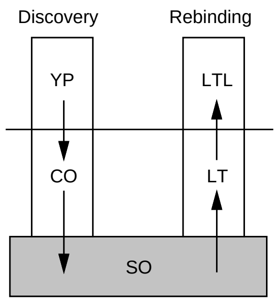

Fig. 3. *Discovery and rebinding are complementary parts of the location service.*

- **Yellow Pages (YP):** The Yellow Pages is the Nomad URN resolver, returned by the URN RDS from Fig. 2. The Yellow Pages takes a textual URN string and returns a reference to a 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):** A required part of each application within Nomad. A location table contains references to all of the Clusters composing an application and is responsible for tracking their movement.
- **Location Table Locator (LTL):** The location table locator is a well known Nomad service. It is responsible for locating and tracking all the location tables of all of the applications executing within Nomad. The initial reference to the LTL is obtained via the Yellow Pages and cached by each Depot.
- **Local Interface Repository (LIR):** This is not part of the location service as such (not shown in Fig. 3), 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.

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 in a mobile Cluster share the same location table.

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 Yellow Pages 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.

When an object moves, all existing bindings held on that object become outdated. Resolving these bindings needs the rebinding hierarchy of the location service. Once a binding is outdated, subsequent invocations on that object will fail. The local proxy representing the binding will transparently rebind via a cached reference to 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 the next level — the location table locator (LTL).

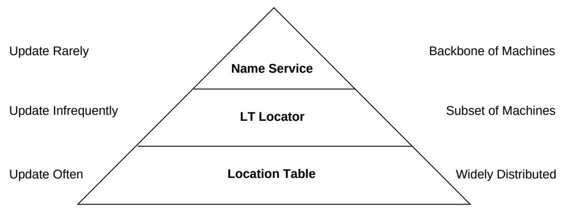

Fig. 4. *Division of Labour.*

Figure 4 illustrates the division of labour between the components of the naming and location service resulting from this location service architecture. As the load increases towards the bottom of the pyramid, so does the degree of distribution.

The remainder of this paper is dedicated to discussing in greater detail, the individual location service components. Sections 3.4 through 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.

3.4 Creating an Application

Let's first consider creating 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 Fig. 5. The vHost starts by creating a Cluster for holding the new application's location table, and sets the resolver[1] for the location table Cluster to be the location table locator. The location table object is then instantiated inside the new Cluster.

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

[1] Each mobile Cluster has a cached reference to its resolver.

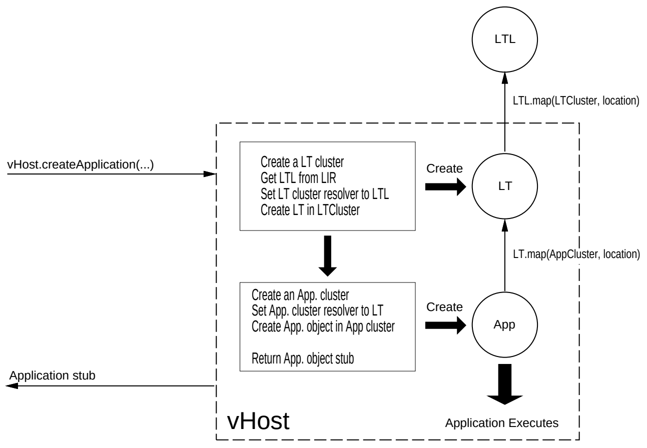
LTL
LTL.map(LTCluster, location)
vHost.createApplication(...)
Create a LT cluster
Get LTL from LIR
Set LT cluster resolver to LTL
Create LT in LTCluster
Create
LT
LT.map(AppCluster, location)
Create an App. cluster
Set App. cluster resolver to LT
Create App. object in App cluster
Return App. object stub
Create
App
Application stub
vHost
Application Executes

Fig. 5. *Creating an application in Nomad*

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

3.5 Registering a Service

The creation of the application will not automatically create a corresponding entry in the Yellow Pages. It is now up to the application to register a URN via the steps illustrated in Fig. 6. Otherwise it can simply remain anonymous to the outside world and respond only to the holder of the interface returned upon its creation.

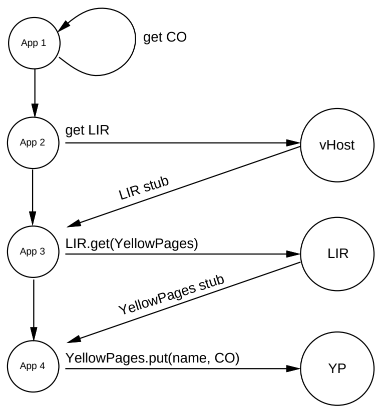

Fig. 6. *Registering a service*

The application starts by obtaining a reference to a contact object (CO) representing itself, usually by creating it. The next step is to find the Yellow Pages with which the application needs to register, by using the Depot's LIR to acquire a reference to the Yellow Pages. With the Yellow Pages 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, and will return the appropriate service object for each level. Alternatively the application could register three different URNs, each with a unique contact object for the individual service levels.

3.6 Obtaining a Service

Figure 7 illustrates the four steps and seven messages which are needed to obtain a service from an application, 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 Yellow Pages is available from the LIR, the external access is shown here for completeness.

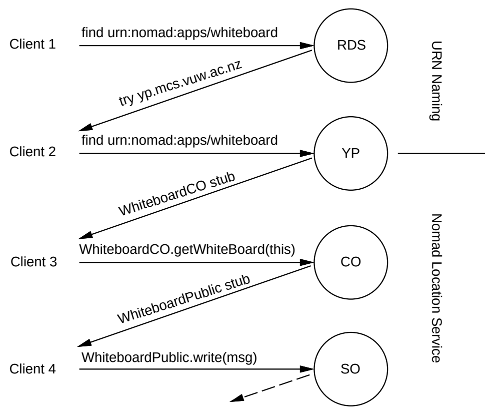

Fig. 7. *Locating an Object.*

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 Yellow Pages, to which a reference is returned by the RDS. The client then queries the Yellow Pages 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.

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 Sect. 3.7.

3.7 Moving and Rebinding an Object

From the perspective of the location service, moving an existing object is similar to creating a new object. The migration process transparently invokes the map method on the location table responsible for tracking the migrating object's Cluster. This ensures the correct binding of future references, but does not resolve the issue of outstanding references which no longer specify the correct location.

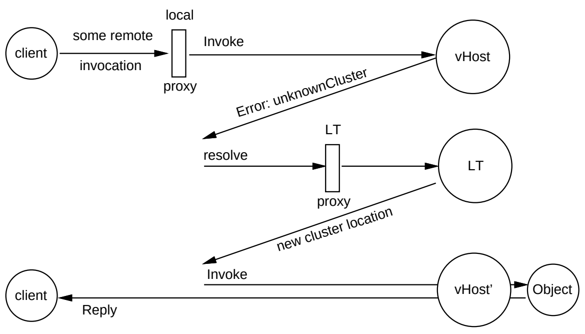

Fig. 8. *Rebinding to an Object.*

Fig. 8 illustrates what happens when one such outdated reference is used to invoke a method on a remote object. The invocation by the client causes the local proxy to use the object reference it held from the previous invocation. The proxy contacts a vHost which no longer hosts the target object. When the vHost's communications stack attempts to invoke the remote method on the object an exception occurs and an error is returned to the invoking proxy.

The local proxy next uses a cached reference to the location table responsible for tracking the target object's Cluster to request a new binding. The 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 the call to the location table is simply another remote invocation via a proxy. Since the location table is also mobile, this may result in a cascaded rebinding, as shown in Fig. 9.

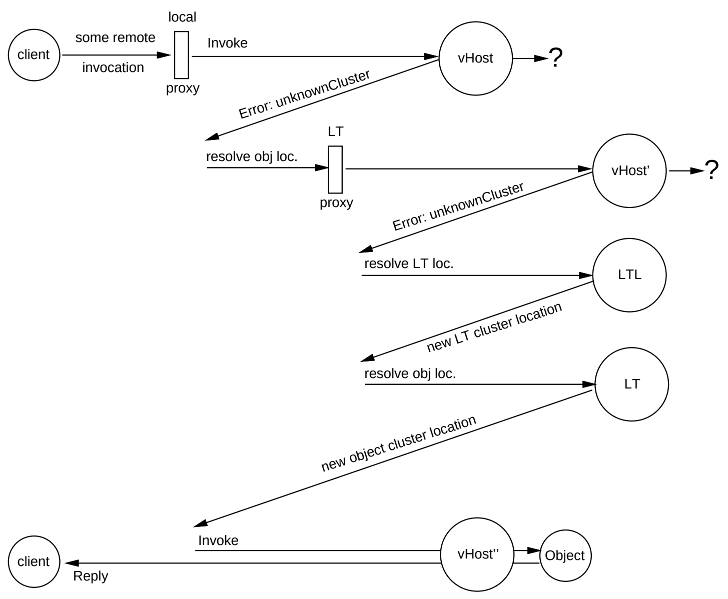

Fig. 9. *Cascaded location table rebinding.*

In Fig. 9, the call to the location table by the location table proxy fails. As in Fig. 8, this results in an error being returned to the location table proxy. The location table proxy then acts in exactly the same way as the local proxy 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 location table's location is updated within the proxy, and the original query reissued to the location table. The location table returns the new location of the target object's Cluster to the original proxy, and the original invocation is reissued.

In the two previous examples all rebinding took place dynamically and transparently to the application. It is worth summarising a few cases where transparency can no longer be maintained:

– In the case where an **object no longer exists**, the same steps as shown in Fig. 8 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 is application dependent.
– 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 from Fig. 7 need to be repeated.

A significant advantage of the separate rebinding hierarchy is now evident — it does not matter if the contact object references held by the Yellow Pages 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 Yellow Pages does not need to be immediately updated to reflect changes due to mobility of the contact objects. Instead, the Yellow Pages can utilise a lazy, best effort update as omissions, delays and inconsistencies will be caught and then corrected by the rebinding system.

4 Implementation

The location service has been implemented in a Java prototype of the Nomad architecture. Location tables are created and managed automatically as part of the Nomad mobile Cluster package. The system accommodates transparent replication of location tables, consistency and fault tolerance, details of which appear in [2].

This paper has concentrated on the overall architecture of the twin discovery and resolution hierarchies, and the novel application level location tables. Development of a specific location table location service would be redundant, as other research efforts, in particular Globe [5–7], are readily applicable solutions for this component of the system.

5 Summary

This paper presents a novel solution to the design of a distributed location service for large scale mobile object systems. The use of the application to optimise the distribution of the location tables, limits the impact of the majority of updates to these small and infrequently mobile data structures. The remaining global workload only involves resolving the locations of the location tables themselves, ensuring an implicit ability to scale. Higher level bindings are stable within the discovery hierarchy.

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

References

1. Lange, D.B., Oshima, M.: Seven Good Reasons for Mobile Agents. Communications of the ACM **42** (1999) 88–89
2. Bubendorfer, K.: NOMAD: Towards an Architecture for Mobility in Large Scale Distributed Systems. PhD thesis, Victoria University of Wellington (2001)
3. Mockapetris, P.: Domain Names: Concepts and Facilities. Network Working Group, Request for Comments (RFC) 1034 (1987)
4. Comer, D.E.: Computer Networks and Internets. 1st edn. Prentice-Hall (1997)

5. Ballintijn, G., van Steen, M., Tanenbaum, A.S.: Exploiting Location Awareness for Scalable Location-Independent Object IDs. In: Proceedings of the Fifth Annual ASCI Conference, Heijen, The Netherlands, Delft University of Technology (1999) 321–328
6. van Steen, M., Homburg, P., Tanenbaum, A.S.: Globe: A Wide-Area Distributed System. IEEE Concurrency **7** (1999) 70–78
7. van Steen, M., Hauck, F.J., Homburg, P., Tanenbaum, A.S.: Locating Objects in Wide-Area Systems. IEEE Communications Magazine **36** (1998) 104–109
8. Tanenbaum, A.S.: Computer Networks. 3rd edn. Prentice-Hall (1996)
9. Damianou, N., Dulay, N., Lupu, E., Sloman, M.: 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)
10. Bubendorfer, K., Hine, J.H.: DepotNet: Support for Distributed Applications. In: Proceedings of INET'99, Internet Society's 9th Annual Networking Conference. (1999)
11. Vickrey, W.: Counterspeculation, Auctions, and Competitive Sealed Tenders. The Journal of Finance **16** (1961) 8–37
12. Sollins, K., Masinter, L.: Functional Recommendations for Internet Resource Names. Network Working Group, Request for Comments (RFC) 1737 (1994)
13. Sollins, K.R.: Requirements and a Framework for URN Resolution Systems. Internet Engineering Task Force (IETF) Internet-Draft (1997)
14. Slottow, E.C.: Engineering a Global Resolution Service. Master's thesis, Laboratory for Computer Science, Massachusetts Institute of Technology (1997)
15. Moats, R.: URN Syntax. Network Working Group, Request for Comments (RFC) 2141 (1997)
16. Iannella, R., Sue, H., Leong, D.: BURNS: Basic URN Service Resolution for the Internet. In: Proceedings of the Asia-Pacific World Wide Web Conference, Beijing and Hong Kong (1996)