A Managed Architecture for Mobile Distributed Applications

T. Walsh, P.A Nixon, S.A Dobson

Computer Science Department,
University of Dublin,
Trinity College, Dublin 2, Ireland.
Tim.Walsh@cs.tcd.ie

Abstract: Internet-distributed systems are beginning to offer a serious platform for stable, long-lived, flexible applications development. Moreover, such Internet-scale applications are motivating new modes of work and company integration such as teleworking and virtual enterprises. There is an increasing realisation that within such flexible working structures that applications will have to support some form of mobility, both to handle explicitly mobile nodes (such as users with lap-tops) and to provide reconfiguration and redeployment of application components around the Internet. A major challenge for such mobile distributed applications is to provide stability in the face of constant change, providing application developers and users alike with a stable architecture whose elements and locations may change across time. This paper presents an architecture that separates location and mobility knowledge from algorithmic structure within an application, thus facilitating a managed solution to adaptability and mobility.

Keywords: migratory applications, distributed systems, mobility, software architecture.

1   Introduction

Internet-distributed systems are beginning to offer a serious platform for stable, long-lived, flexible applications development. There is an increasing realisation that such applications will have to support some form of mobility, both to handle explicitly mobile nodes (such as users with lap-tops) and to provide reconfiguration and redeployment of application components around the Internet. The scale and dynamism of this type of Internet application is exemplified in the ideas of teleworking or virtual enterprises. In these situations individual users or corporations must interact with rapidly changing collections of other workers or corporations in a secure and flexible manner. Issues such as the current location of the worker, trustworthiness of the worker, and access to data must be managed in addition to the normal function of the application being utilised. A major challenge for such distributed applications is to provide stability in the face of constant change, providing application developers with a stable architecture whose elements and locations may change across time.

We are particularly concerned with building *virtual enterprises* (Internet-scale mission-critical distributed systems spanning a changing number of component organisations and administrative domains) because such systems are the essential infrastructure for projects ranging from electronic commerce to large-scale scientific collaborations. Although the essential low-level technologies exist, issues such as component selection, re-use, re-configuration, trust, authentication, performance, efficiency and robustness remain to be addressed in a way which gains sufficient management trust to bring the technology to fruition.

Current solutions to decentralised applications alone will not service the needs of this type of highly dynamic and mobile application. What is required is an inherently

mobile solution detached from the present client-server model and the associated problems of scalability and flexibility [1]. These solutions will exploit mobile code because of its capability to dynamically change the bindings between code fragments and the location where they are executed [2]. This enables a migrating application to dynamically rebind to a generically named resource on a destination host while having the option of maintaining links to previously visited hosts.

In this paper we present a design strategy for managed mobile distributed applications in which we clarify the relationship between algorithm and location management. The motivation for this separation is the same as for the implementers of client/server systems such as CORBA, namely to implement network operations transparently. In this case the task is to implement the mobility aspect in a transparent manner thus allowing a single algorithm to work within a range of mobility policies. We derive an architecture for applications using this design strategy which supports the integration of a range of mobile programming paradigms within an open framework.

Section two discusses the characteristics of mobile distributed. Section three presents the design strategy, which is elaborated into an open architecture in section four. Section five explores programming within such an architecture, whilst section six contrasts the approach with other systems from the literature. Section seven concludes with some directions for future work.

2   Mobile Distributed Applications

Code that executes on the majority of machines today may be classed as static code because it remains on a single node for the duration of its life. In terms of execution speed it is the most efficient type of code, but it is not designed to ease communication with other nodes. In response to this observation, designers created the client-server paradigm [13] to allow executing processes on one node to avail of services on another, typically more powerful, machine. This concept was taken a stage further with peer to peer networking which enabled any machine to temporarily act as a server while communicating with client requests [6].

A logical advance of this concept was to free the actual processes from the confines of their nodes and allow the code to move to different nodes rather than just having remote method invocation. This approach is seen primarily in two distinct domains: intelligent agents [5] and process migration [14]. The concept of moving computational 'know-how' in the form of code across nodes gives some distinct advantages over more static techniques. The advantages of migration include [3];

- Load sharing – By moving objects around the system, one can take advantage of underused processors

- Communication performance – active objects which interact intensively can be moved to the same node to reduce the communication cost for the duration of their interaction

- Availability – Objects can be moved to different nodes to improve the service and provide better failure coverage.

- Reconfiguration – Migrating objects permit continued service during upgrade or node failure.

- Location Independence – An object visiting a node can rebind to generic services without needing to specifically locate them.

The key concept in this type of system is its dynamic adaptability; the capability to conform to different situations at runtime. Systems such as these, despite having a simple concept, present a major challenge in actual implementation. A range of difficult dilemmas manifest themselves including servicing re-binding, administering the binding type and maintaining the system.

Prior to migration an object can have bindings to many other different services and resources. Examples include printers, displays and databases. After migration some bindings will remain the same while others may require rebinding to similar typed resources on the new node. Thus to perform tasks and integrate successfully with applications on a given system objects must bind to resources at particular nodes and this dynamic re-binding must be automatic and transparent.

2.1   Binding

Foundational to the operation of mobility in distributed applications is binding. In [1] binding is described as the act of attaching to a resource by means of an appropriate type of reference. Such bindings are characterised by the fact they are transferable or not transferable, and by the strength of the binding (reference).

A binding by 'identifier' is the strongest type. In this case the resource is unique and cannot be replicated. Such bindings remain throughout an object's lifetime therefore any migration by the object will retain a network reference to the original resource unless the resource in question is transferable. In this case the resource could also be moved but bindings from other objects would need to be subsequently updated.

A binding by 'value' is weaker than by identifier. Such bindings declare that, at a given moment, the resource must be compliant with a given type and its value cannot change as a consequence of migration. This kind of binding is usually exploited when an object is interested in the contents of a resource and wants to be able to access them locally. In this instance the resource could be moved with the object if it is a transferable resource or bound by network reference if the resource is fixed.

A binding by 'type' is the weakest or the types. Such resources are generic resources that are typically available at any node. Examples include printers and displays. This type of binding is re-bound to the local resource on the new node. The management of such resources will be subject to the rules of the architecture. Binding types can increase in strength but never decrease. For example, if an object changes a database resource then that database has now gone from a duplicated resource to a unique resource and so the binding would need to be changed to an identifier type. This represents the resource's new uniqueness.

Separating the algorithm from its location will require a system that can provide resources transparently using a combination of services such as naming, trading and

relationship [4]. At one level of abstraction each of these services will simply be managing binding. The benefit of abstracting a number of behaviours and managing them by re-binding, is that the algorithm can retain consistent access to a resource whose reference is managed by some external services. So the need for the programmer to explicitly cater for rebinding is now devolved to an external service. It is worth noting that the service can apply policies to the binding process to enforce a user or enterprise requirement such as security or performance policies. As an example, the maintenance of a distributed system involves all the usual tasks of incorporating performance tuning, system upgrades and fault tolerance. Since the system is not centralised it should be possible for one node to be taken down to allow upgrades to be performed without compromising the remainder of the system. Most, if not all, of these changes can be addressed by the introduction of our managed architecture. This particularly pays off in large-scale Internet systems where, because of their peer-to-peer nature, control and maintenance can become problematic.

3   Design Strategy

In this section we explore some of the issues in designing information systems for the virtual enterprise. In particular we elaborate on the evolutionary nature of such an enterprise demonstrating the need for a managed solution. In parallel with this we explore the design strategies which must be employed and identify policy issues which must be captured during design and enforced in the implementation.

3.1   Evolving to the virtual enterprise

A virtual enterprise is a highly dynamic system whose component organisations may change repeatedly over its potentially unbounded lifetime. The information system must react to the changing relationships between the constituent organisations so that (for example) new members receive information while old members do not. These reactions will be controlled by business-level goals and policies, which must be translated reliably into the behaviour of the information system. Issues such as system re-configuration and component deployment do not occur in a technical vacuum but are instead affected directly by business dynamics.

Mission-critical systems tend to be built from tried and trusted components including legacy systems (often with object-oriented wrappers), commodity off-the-shelf applications, and "bespoke" objects specific to the particular business process. This is a trend that should be encouraged: although it makes system integration more complex, it can radically reduce the complexity and novelty of the system and so improve the chances of success. The implication is that a virtual enterprise infrastructure must work with a range of existing methods and technologies, and cannot assume that the new system is being implemented in a "green field".

Recognising the need for a dynamic enterprise, and committing to it, have an interesting side effect. An organisation by definition commits to progressively upgrading its information systems, and may therefore retain legacy systems in an environment which minimises the impact of any necessary changes down the line.

3.2   Separating form and function

Distributed object systems such as CORBA, DCOM and RMI present an abstraction in which objects may interact regardless of their locations. This allows the system designer to locate objects according to whatever high-level criteria are appropriate without complicating the application's code. However the locations are fixed at object creation - once located, an object does not migrate - which is a considerable handicap, preventing a long-lived system from adapting to changing conditions by re-locating services.

Operating systems research in this area has concentrated on process migration, moving the complete execution context of a running process from one machine to another. The great advantage of this technique is that the process is suspended during migration, completely masking the change of location. This advantage is also a critical disadvantage for systems in which location is a factor in acquiring resources, since the process will be unaware the its previous bindings have been rendered obsolete.

At the other extreme, several authors have suggested agent communities as an approach to virtual enterprises. Agents encapsulate algorithmic functionality with control of their own mobility, allowing them to wander around a system performing work locally wherever possible. (In some agent systems there is no remote communication, so all agent interaction occurs locally after migration to a common point.) Although such decentralised control improves communication efficiency, local decision-making is known to have severe drawbacks in complex systems. The combination of locally optimal solutions is often highly sub-optimal, and the interactions of several different control policies (by agents pursuing different goals) rapidly becomes impossible to analyse or predict.

A related problem is the coupling of the function an agent performs and the location at which it performs that function. It can be argued that this coupling stems from the object-oriented principle of complete encapsulation, allowing an agent to be treated as a "black box". However, the alternative view is that this coupling reduces the ability to re-use the agent in varying circumstances, since it may be difficult to change its movement logic while re-using its functionality.

While agent systems undoubtedly have a place in the virtual enterprise (and especially as information gatherers outside the system), we feel that they cannot as yet be recommended for mission-critical infrastructural tasks. We believe that the best approach for virtual enterprise systems is to separate the algorithmic logic of a component from the logic used to locate and re-locate components - essentially divorcing the function of an application from its form over the network. This approach is already used tacitly by components assuming a fixed location: it may be generalised to include components which allow themselves to be migrated by an external entity.

The separate configuration logic can be handled in a number of ways, for example by providing plug-in location managers for components. This would still suffer the

problems inherent in local decision-making. A completely centralised policy manager is superficially attractive but would become a severe bottleneck for large and/or highly dynamic applications. A combination approach is to use a location management component which is itself completely stateless (and which may therefore be replicated freely) but which references other information sources in making its decisions. By pushing the problem back, we gain the ability to use centralised, federated or fully distributed information sources without affecting the design or interactions of components.

The use of a separate configuration component raises a number of other design questions. Chief among these is the way in which re-configuration is seen by components. Can a component request to be moved to a specified location (the agent approach)? Can a component be forced to move, and what are the implications of this on resource binding and concurrency control? Can a component prevent migration, and what does this imply for system predictability? Different scenarios will give different answers to these questions, all or any of which might be desirable. By divorcing application from policy decisions in our architecture we introduce the flexibility to choose the most desirable behaviour for the system at any given time.

3.3 Stability in the face of change

The "software crisis" has shown that current software engineering practices are often less than adequate at handling relative fixed systems. Dynamically adaptive systems therefore offer a severe challenge to the design of systems, programming languages and components. The essential problem is to provide the system designer, system manager and component implementor with a platform that is sufficiently flexible to capture the range of re-configuration operations while retaining sufficient stability to be usable in practice.

We may observe that most systems' architectures change considerably more slowly than their objects. That is to say, a role in a system is longer-lived than the particular object that fulfils it. Thus the distributed object approach of interacting with objects whose names are explicitly known is problematic in the face of long-lived applications.

System architecture is usually considered as a precursor to design rather than a run-time concept. If we maintain a machine-readable description of an application's roles and relationships, we may make high-level role information available directly to components. This would allow, for example, a component to construct a reference to "the user database" as an architectural concept, regardless (at least in some senses) of the database object providing that service.

While the most obvious application of this approach is in providing high-level binding, it may also be applied to location-sensitive resource acquisition if the architecture provides a relationship between a resource and its physical location (which may change across time). This allows a component to specify a binding to a local resource - "the local printer", for example - and have that binding remain valid in the face of changes in location. Thus the same technique can be used to provide both stability in the face of change and sensitivity to local conditions. Several issues arise immediately from this approach.

The first concerns role changes, which must be propagated to all components making use of that role. One may adopt an eager approach (sending notifications to all affected components) or a more lazy approach (forcing component to re-acquire the correct object before use). Both approaches are pathological under some circumstances.

A second issue involves the effects of migration, which may invalidate a specification even if the roles themselves are unaffected. This requires some mechanism in the underlying run-time system to detect location changes, even if location changes are masked from the component in other ways.

The third issue concerns the way in which a component interacts with a resource specified by role. In most current systems the resource will still at some level be represented by explicitly known references, which will be needed for any interaction to occur. We must therefore address the manner in which these names are unpacked from their role-based specification. We must also address the concurrency control issues implicit in changing the object fulfilling a role while the reference is unpacked.

4 A Managed Architecture Mobile Distributed Applications

The structure of a generic migratory distributed application is displayed in figure 4.1. It shows how each component (also referred to as an autonomous object) can exist on a different node and yet remain in communication with other components of the application. Here the term node refers to a single processor, typically a single host. The circles represent components, while the dashed lines represent the various network references, or simply links, between the them. The components are capable of being migrated to another node yet still carry on its computation, and more importantly its role in the computation. This is the goal for a real world system but prior to building such a system it is necessary to describe a disciplined abstract structure.

The complete structure of the architecture can be encapsuated in a logical database. It represents a rational method to store such a configuration because it is well structured, secure and retains data integrity automatically. The logical database can implemented as a single, distributed or federated database, and we use the central database term solely to convey the methodology to be used in implementing such a system.

Different interfaces to the database allow access to the data. These interfaces are in the form of the standard services such as those specified by the OMG, RM-ODP, ODMG, and IETF. Due to the fact that they are stateless they represent logical filters to all data. For consistency we assume the definitions of the OMG for discussions of services named below.
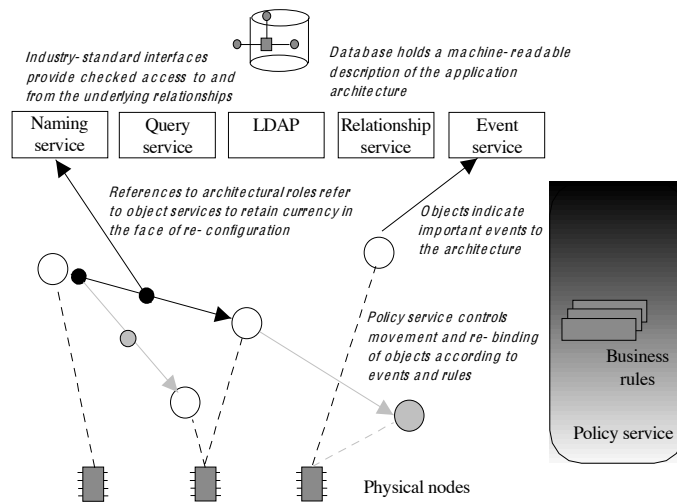
Industry-standard interfaces
provide checked access to and
from the underlying relationships

Database holds a machine-readable
description of the application
architecture

| Naming service | Query service | LDAP | Relationship service | Event service |

References to architectural roles refer
to object services to retain currency in
the face of re-configuration

Objects indicate
important events to
the architecture

Policy service controls
movement and re-binding
of objects according to
events and rules

Business
rules

Policy service

Physical nodes

Figure 4.1 A Managed Architecture Mobile Distributed Applications

The relationship service and naming service will assist migrating components to rebind to well specified or generic services. For example, should a migrating component wish to avail of a standard printing service on a destination host then the relationship service will provide the appropriate resource name. If the docking component has special 'secure' privileges then the relationship service may specify a secure output device such as the manager's printer while 'normal' components are referred to a standard office printer. Using this evaluated resource name a concrete binding to the particular resource is returned from the naming service. The Event Service [4] allows objects to communicate using decoupled event-based semantics. The Query Service provides an interface in much the same way SQL an interface to databases.

All the services see the same data in the database. Their purpose is to give the component a view of just the data needed to provide the bindings between resource names and their object references. This provides a seperation of data from the interfaces used to access it. Migrating components use the service interfaces to reconfigure their resouce bindings. Each migration is different and can have different consequences. Use of standards allow the architecture to evolve as more interfaces become available over time and therefore, such an open system has the potential to do for migratory applications what OMG's CORBA did for remote object invocation.

An important part of the architecture resides in the policy management unit. Each component which is capable of migrating has a 'moveTo()' method for such a purpose. The component itself can not call this method. Instead the policy unit invokes it, providing it with a destination and any other necessary paramenters. The policy manager can dictate for example, whether it will allow a particular component access to its node or even whether an object can leave its node. Such decisions are dynamic and complex, and the design of a comprehensive policy manager is outside the scope of this paper. Were the component able to invoke moveTo() itself it would exibit the primary trait of an autonomous agent. Such autonomy can be simulated within the architecture by the use of a null policy which simply defers to requests of

the requesting object. Policies are the key to adapability, as they also allow a range of different design choices to be facilitated in a single implementation. Policies even enable for design decisions to be revised post implementation without the necessarily having to change the algorithmic solution.

Central to a stable and smooth migration is the concept we refer to as an *architectural pointer*: a reference to an object specified relative to the architecture database by way of one of more of the services made available by the architecture. The pointer essentially encapsulates a query against the database together with the object resulting from the last time the query executed.

The power of architectural pointers comes from the use of the database. By combining architectural, configuration and placement data within a single logical structure, and allowing different views through the services, it is possible to identify roles in a very flexible manner. It also allows objects to reference roles directly, rather than purely retaining links to the objects which happened to fulfill those roles at any point in time. Finally, by acting as event consumers, they allow the policy component to re-configure the application and propagate these changes (in the form of pointer invalidations) directly to all affected objects.
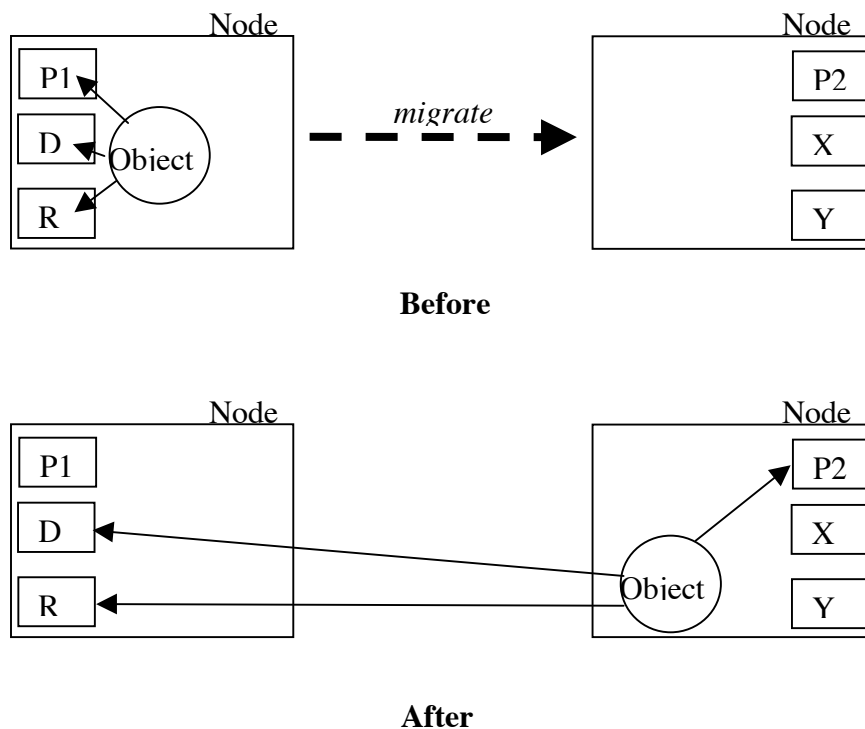

Before

After

Figure 4.2 Architectural Pointers

As an example of the power of architectural pointers consider an object using both shared user database (D2) and a printer (P1), with the constraint that the object always uses the same database irrespective of location and the printer associated with its local node. Architecturally this may be specified by identifying the user database explicitly by name and the printer in relation to the local node (figure 4.2 - before). If the object is migrated (figure 4.2 - after) then the database reference will remain valid but the printer reference will be invalidated, and when re-evaluated will re-bind to the printer

on the object's new node (P2). The invalidation in this case is performed by the migration run-time system.

5   System Integration and Management

The deployment of a distributed architecture which allows mobility can be viewed from the perspective of the management of the system, the design of applications, the relationship of code to architecture, and from the standpoint of security. The importance of taking this approach is justified by the separation of concerns for the various groups of people who will use the system. An administer running the system will not be concerned with the aspects relating to programmers, for instance.

5.1   Management View

We take a top down view of the management of the system. It is primarily concerned with the single (albeit distributed) view of an application. The actual mechanics of migration pertain to an application view of the system that is discussed in section 5.2.

The system is built from components that are its basic unit of construction. They run in a computation environment provided by each host. This is same approach taken when applets (components) are downloaded to the Java Virtual Machine (computation environment). From a management point of view, control of the computational environment means controlling the system. This differs from an intelligent agent [5] based system that is, more or less, self-controlling because of the agent's learning capabilities and artificial intelligence characteristics.

The policy management unit dictates the rules to be followed on the individual node. The policy decides whether a migrating component will be allowed to execute on the target host. This aspect lends itself to toward the security issues that are discussed in section 5.4. The node, as mentioned in section 4, is responsible for hosting of migrating components. The main task is the examination of the component bindings and subsequent evaluation of how to handle them. The generic resources which components bind to on nodes are easily substituted on the new node. Using the procedures for re-binding the component requests new CORBA style object references from the naming service.

The administrator of a node will want control over what resources are available. The various service interfaces in the form of the naming, relationship, query and event service require an authority to update and maintain. This is the basic role in management. It administers any new entries to the logical database of data. The advanced role of the administrator is that of policy making.

Moreover, as the system evolves it is necessary to maintain backward compatibility by retaining a versioning and architectural history of the system. The consequence of ignoring previous versions will be a fragmentation of the architecture. Application of policies at this level can ensure no decisions are taken which introduce compatibility problems. Critically all these management views mean that the relationship, **business goal** implies **policy** implies **adaptation**, can be achieved more readily.

5.2　Application View

The application view is a component view of the system. Although a management view of the system is essentially a large single logical database the reality is that each node will service its own database which retains the node's state and architectural pointers.

Each application is made up of components all doing various tasks, some more important that other. At its most basic, the component is a set of instructions that need to be carried out on predefined objects. It is analogous to a drone engineer who needs to inspect several key sites. The engineer is transported to a site, performs its tasks and via its mobile phone, a link back to resources on other 'nodes' such as its secretary back at head office, is informed of where the next job will be.

When a component moves to another node it needs to need to bring two primary subcomponents [1]; the code segment which provides the static description for the behaviour of a computation and the state composed of a data space and an execution state. The data source is the set of references to resources that can be accessed by the component. The execution state contains private data that cannot be shared, as well as control information related to the component's state, such as the call stack and instruction pointer.

Once the migration has occurred there may be a set of bindings to resources, which the programmer has referenced, which can be replaced by a similar resource on the new node. A process of tracking through these bindings and automatically rebinding to new, equivalent resources, enables the component to migrate transparently. Once all bindings have been sorted the component can resume execution. Network references to irreplaceable, unique resources remain as they are. Because some components migration may result in poor performance the component may, to ease network traffic, be returned to the original node. This will depend on the policy manager on the node that is hosting the component. The programmer may set a granularity level which describes the resistance the component will exert against a request by a node to move it elsewhere. Obviously the node will have the final say in whether the component is moved. Problems arising because the host can not find any other hosts willing to accept the component will, dependant on policy, result in the component being placed in a suspended state.

5.3　Relationship of code to architecture

Various issues need to be examined in terms of migration control. Questions arise as to how much autonomy components should have. If the code can move itself there are policy issues to examine, as have been investigated in the agent community. It may not be possible for a certain agent to move to a node due to size or security restrictions. This is the primary reason why agents have portfolios. The portfolios are similar to the briefcase notion used by Cardelli in Visual Obliq [7]. These portfolios are forwarded to the destination nodes prior to the actual migration and checked by policy management units for legality. If all requirements are fulfilled then the code can migrate.

The problem arises when the architecture tries to optimise the system. For example, if a component is running on a Node A yet accessing a resource on Node B it may be advantageous to also move the resource to Node A. If this resource is a standard resource is should be able to simply copy it. If however, the resource is unique it will have to be moved. This move will include all the present references to it from various other components. This could in turn start a pathological cascade of moves in other parts of the system.

One of the ancillary motivations for our managed solution is to investigate these interactions and report appropriate policy patterns that can be used to avoid such situations. If the architecture is solely responsible for initiating moves then communication among the participating nodes can foresee such problems and avoid the initial problems. On a small set of nodes the problem is controllable, but when scaling systems up to include many thousands of nodes it is likely that such behaviour would lead to chaos with no realistic way of maintaining logs at to where everything is and where everything is going.

If the architecture moves the code then the principal difficulty is in concurrency control and how to maintain it. The system would require separation of concerns using critical sections and rollbacks using well defined ACID techniques [6].

5.4   Security Issues

Since the system is basically a logical database the security issues have been largely address in the database world. Problems relating to concurrency, security and integrity are addressed. These solutions are largely to deal with problems in invalid data accessing. The other area of major concern is that of malicious attack. Since migrating components have the services of the host computer at their disposal it leaves many vulnerable systems open to attack. These are essentially problems of authentication and trust. Using public/private key technology visiting components can receive various trust levels and the corresponding extra access. This leads to the concept of trust webs and evolving trust [15].

Prior to accepting a component its portfolio is examined and from its list of bindings it may be desirable to disallow access to components that have binding to insecure and entrusted nodes. This is a policy issue once again which is driven by some low-level identification mechanism for objects and location.

6   Related Work

There are several bodies of work that provide solutions to key aspects of a mobile distributed architecture. A number of key ones are described below with further discussion found in [1,11,12].

In Emerald [9] an object can be moved to a new location specified by node name or another object location. Emerald also allows global objects, which are objects that can be referenced from anywhere in the distributed system, can be accessed remotely, and can move independently. A feature of the design of Emerald is that it requires an explicit notion of object location and movement is under explicit programmer control.

Emerald was limited to homogeneous local area networks. Both of the transparency and the homogeneity issues are overcome with our architecture.

Obliq [10] allows for the remote execution of procedures by means of execution engines that run on the host. An Obliq thread can request the execution of a procedure on a remote node. The shipping of code is synchronous and mobility never changes bindings, leading to an object model with distributed scope.

ObjectSpace's Voyager implements object migration using Java technology but does not provide policy management options. It has a utility which takes any Java class and creates from it, a remotely accessible equivalent, called a virtual class. Voyager can create an instance of a virtual class on a remote host, resulting in a *virtual reference* that provides location independent access to the instance. Programmers use this mechanism for implementing classes.

IBM's Aglets is a fully Java compliant agent system. The agent, or aglet, migration is absolute (a server name is specified) with no links kept to the previous system. Message passing is the only mode of communication supported as aglets cannot invoke each others' methods. Messages are tagged objects and can be synchronous, one-way or future reply. The system provides a *retract* primitive that recalls and aglet to the caller's server, but there is no access control on this primitive.

Telescript is a commercial mobile agent system from General Magic. Telescript servers, called *places*, offer services to visiting agents in much the same way we plan to. Our system rebinds to services that are similar to services on a previous host automatically as opposed to Telescript's 'service station' type concept.

While these systems implement key aspects of a mobile distributed architecture, they do not provide:

1. any policy management options;
2. any mechanism for automatically re-binding to other resources on the destination node; or
3. transparency of policy application.

These three issues are the main driving forces behind the proposed system; to integrate distributed systems with a serviced policy manager controlling the migration process.

7   Conclusions

The trend towards virtual enterprises demands a new focus within systems design, away from simple static composition and towards managed adaptability responsive to changing business goals and conditions. The key aspect of this new focus is the ability to handle migration and replication of both users and software components. A clear and well-founded approach to these issues is imperative if computer systems are to keep up with the new levels of business reactivity made possible by Internet-enabled commerce.

In this paper we have set out a design strategy for building highly adaptable mission-critical systems distributed at Internet scales. This strategy emphasises the clear separation of algorithm from configuration management, and highlighted some problems that arise in implementation as a result. We used this analysis as a guide to construct a managed architecture for adaptable systems. The architecture synthesises a machine-readable architectural description accessed through a set of industry-standard interfaces. All configuration management logic is encapsulated into a replicable policy component responsible for adapting the application to changing conditions according to business rules. Application components access architectural roles directly using a system of architectural pointers responding directly to configuration changes.

We have performed a number of small-scale experiments with both the design strategy and the techniques described, with very promising results in terms of the ability of the architecture to mask awkward and error-prone aspects of adaptive behaviour. These experiments have also shown the feasibility of combining largely location-unaware components with a managed infrastructure. Our next set of experiments will seek to show the practical application of these techniques to realistically sized enterprise information systems to explore the trade-offs inherent in managing the information infrastructure for a highly dynamic and distributed virtual organisation.

8   References

[1] A. Fuggetta, G.P. Picco, G. Vigna, "Understanding Code Mobility" IEEE Trans of Software Eng., vol 24, no. 5, pp. 342-361, May 1998.

[2] A. Carzaniga, G.P. Picco, and G. Vigna, "Designing Distributed Applications with Mobile Code Paradigms," Proc. 19th Conf. Software Eng. (ICSE'97)

[3] E. Jul, H. Levy, N. Hutchinson, A. Black, "Fine-Grained Mobility in the Emerald System" Acm Trans. On Computer Systems, Vol. 6, no. 1, pp. 109-133, February 1988.

[4] OrbixWeb Programmers Guide. IONA Technologies plc, September 1998

[5] S.Green, L.Hurst, B.Nagle, P.Cunningham, F.Somers, R.Evans, "Software Agents: A Review", Trinity College, Broadcom Eireann Reasearch Ltd.

[6] J. Bacon, "Concurrent systems: an integrated approach to operating systems, database, and distributed systems", Addison-Wesley, 1992

[7] K. A. Bharat, L. Cardelli, "Migratory Applications", Digital SRC research Report 138, 1996

[8] CORBA/IIOP 2.2 Specification, Object Management Group, 1998

[9] A. Black, N.C.Hutchenson, E. Jul, H. Levy, "Object structure in the Emerald system" Proceedings of 1986 Conference on Object-Oriented Programming Systems, Languages and applications, Ass. of Computer Machinery.

[10] L.Cardelli, "A language with Distributed Scope", Computing Systems, vol 8, no.1 , pp. 27-59, 1995

[11] N. M. Karnik, A. R.Tripathi, "Design Issues in Mobile-Agent Programming Systems", IEEE Concurrency, Vol 6, No. 3, July – Sept 1998.

[12] V. Cahill, P. Nixon, B. Tangney, and F. Rahbi, "Object Models for Distributed or Persistent Programming", The Computer Journal, Vol. 40, No. 8, 1997.

[13] S. Mullender (Editor), "Distributed Systems", Addison Wesley, pp. 594, 1993.

[14] A. Zamoya (Editor), "Parallel and Distributed Computing Handbook", pp. 1198, 1996.

[15] R. Khare and A. Rifkin. "Trust Management Issues for the World", in Computer Networks and ISDN Systems, Volume 30, Pages 651-653, 1998.