The MultiSpace: an Evolutionary Platform for Infrastructural Services

Steven D. Gribble, Matt Welsh, Eric A. Brewer, and David Culler
The University of California at Berkeley
{gribble,mdw,brewer,culler}@cs.berkeley.edu

Abstract

This paper presents the architecture for a *Base*, a clustered environment for building and executing highly available, scalable, but flexible and adaptable infrastructure services. Our architecture has three organizing principles: addressing all of the difficult service fault-tolerance, availability, and consistency problems in a carefully controlled environment, building that environment out of a collection of execution environments that are receptive to mobile code, and using dynamically generated code to introduce run-time-generated levels of indirection separating clients from services. We present a prototype Java implementation of a Base called the *MultiSpace*, and talk about two applications written on this prototype: the Ninja Jukebox (a cluster based music warehouse), and Keiretsu (an instant messaging service that supports heterogeneous clients). We show that the MultiSpace implementation successfully reduces the complexity of implementing services, and that the platform is conducive to rapid service evolution.

1 Introduction

`The performance and utility of a personal computer will be defined less by faster Intel processors and new Microsoft software and increasingly by Internet services and software.`

c|net news article excerpt, 11/25/98

Once a disorganized collection of data repositories and web pages, the Internet has become a landscape populated with rich, industrial-strength applications. Many businesses and organizations have counterparts on the web: banks, restaurants, stock trading services, communities, and even governments and countries. These applications possess similar properties to traditional utilities such as the telephone network or power grid: they support large and potentially rapidly growing populations, they must be available 24x7, and they must abstract complex engineering behind simple interfaces. We believe that the Internet is evolving towards a service-oriented infrastructure, in which these high quality utility-like applications will be commonplace. Unlike traditional utilities, Internet services tend to rapidly evolve, are typically customizable by the end-user, and may even be composable.

Although today's Internet services are mature, the process of erecting and modifying services is quite immature. Most authors of complex, new services are forced to engineer substantial amounts of custom, service-specific code, largely because of the diversity in the requirements of each service—it is difficult to conceive of a general-purpose, reusable, shrink-wrapped, adequately customizable and extensible service construction product.

Faced with a seemingly inevitable engineering task, authors tend to adopt one of two strategies for adding new services to the Internet landscape:

Inflexible, highly tuned, hand-constructed services: by far, this is the most dominant service construction strategy found on the Internet. Here, service authors carefully design a system targeted towards a specific application and feature set, operating system, and hardware platform. Examples of such systems are large, carrier-class web search engines, portals, and application-specific web sites such as news, stock trading, and shopping sites. The rationale for this approach is sound: it leads to robust and high-performance services. However, the software architectures of these systems are too restrictive; they result in a fixed service that performs a single, rigid function. The large amount of carefully crafted and hand-tuned code means that these services are difficult to evolve; consider, for example, how hard it would be to radically change the behavior of a popular search engine service, or to move the service into a new environment—these sorts of modifications would take massive engineering effort.

"Emergent services" in a world of distributed objects: this strategy is just beginning to become popularized with architectures such as Sun's JINI [31] and the ongoing CORBA effort [25]. In this world, instead of erecting complex, inflexible services, large numbers of components or objects are made available over the wide area, and services emerge through the composition of many such components. This approach has the benefit that adding to the Internet landscape is a much simpler task, since the granularity of contributed components is much smaller. Because of the explicit decomposition of the world into much smaller pieces, it is also simpler to retask or extend services by dropping one set of components and linking in others.

There are significant disadvantages to this approach. As a side-effect of the more evolutionary nature of services, it is difficult to manage the state of the system, as state may be arbitrarily replicated and distributed across the wide area. Wide-area network partitions are commonplace, meaning that it is nearly impossible to provide consistency guarantees while maintaining a reasonable amount of system availability. Furthermore, although it is possible to make incremental, localized changes to the system, it is difficult to make large, global changes because the system components may span many administrative domains.

In this paper, we advocate a third approach. We argue that we can reap many of the benefits of the distributed objects approach while avoiding difficult state management problems by encapsulating services and service state in a carefully controlled environment called a *Base*. To the outside world, a Base provides the appearance and guarantees of a non-distributed, robust, highly-available, high-performance service. Within a Base, services aren't constructed out of brittle, restrictive software architectures, but instead are "grown" out multiple, smaller, reusable components distributed across a workstation cluster [3]. These components may be replicated across many nodes in the cluster for the purposes of fault tolerance and high performance. The Base provides the glue that binds the components together, keeping the state of replicated objects consistent, ensuring that all of the constituent components are available, and distributing traffic across the components in the cluster as necessary.

The rest of this paper discusses the design principles that we advocate for the architecture of a Base (section 2), and presents a preliminary Base implementation called the Ninja MultiSpace[1] (section 3) that uses techniques such as dynamic code generation and code mobility as mechanisms for demonstrating and evaluating our hypotheses of service flexibility, rapid evolution, and robustness under change. While we have begun preliminary explorations into the scalability and high availability aspects of our prototype, that has not been the explicit focus of this initial implementation, and instead remains the subject of future work. Two example services running on our prototype Base are described in section 4. In section 5, we discuss some of the lessons we learned while building our prototype. Section 6 presents related work, and in Section 7 we draw conclusions.

[1]The MultiSpace implementation is available with the Ninja platform release - see `http://ninja.cs.berkeley.edu`.

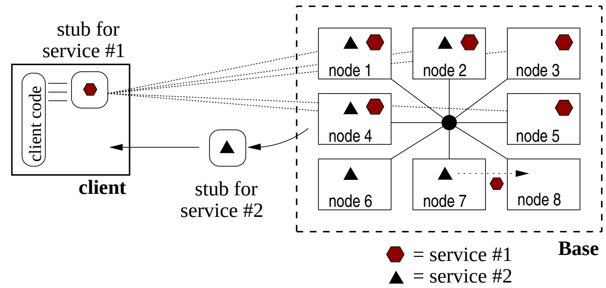

Figure 1: **Architecture of a Base:** a Base is comprised of a cluster of workstations connected by a high-speed network. Each node houses an execution environment into which code can be pushed. Services have many instances within the cluster, but clients are shielded from this by a service "stub" through which they interact with the cluster.

2 Organizing Principles

In this section we present three design principles that guided our service architecture development (shown at a high level in figure 1):

1. Solve the challenging service availability and scalability problems in carefully **controlled environments (Bases)**,
2. Gain service flexibility by decomposing Bases into a number of **receptive execution environments**, and
3. Introduce a level of indirection between the clients and services through the use of **dynamic code generation techniques**.

We now discuss each of these principles in turn.

2.1 Solve Challenging Problems in a Base

The high availability and scalability "utility" requirements that Internet services require are difficult to deliver; our first principle is an attempt to simplify the problem of meeting them by carefully choosing the environment in which we tackle these issues. As in [11], we argue that clusters of workstations provide the best platform on which to build Internet services. Clusters allow incremental scalability through the addition of extra nodes, high availability through replication and failover, and cost-performance by using commodity building blocks as the basis of the computing environment.

Clusters are the backbone of a *Base*. Physically, a Base must include everything necessary to keep a mission-critical cluster running: system administrators, a physically secure machine room, redundant internal networks and external network feeds, UPS systems, and so on. Logically, a cluster-wide software layer provides data consistency, availability, and fault tolerance mechanisms.

The power of locating services inside a Base arises from the assumptions that service authors can now make when designing their services. Communication is fast and local, and network partitions are exceptionally rare. Individual nodes can be forced to be as homogeneous as necessary, and if a node dies, there will always be an identical replacement available. Storage is local, cheap, plentiful, and well-guarded. Finally, everything is under a single domain, simplifying administration.

Nothing outside of the Base should try to duplicate the fault-tolerance or data consistency guarantees of the Base. For example, e-mail clients should not attempt to keep local copies of mail messages except in the capacity of a cache; all messages are permanently kept by an e-mail service in the Base. Because services promise to be highly available, a user can rely on being able to access her email through it while she is network connected.

2.2 Receptive Execution Environments

Internet services are generally built from a complex assortment of resources, including heterogeneous single-CPU and multiprocessor systems, disk arrays, and networks. In many cases these services are constructed by rigidly placing functionality on particular systems and statically partitioning resources and state. This approach represents the view that a service's design and implementation are "sanctified" and must be carefully planned and laid out across the available hardware. In such a regime, there is little tolerance for failures which disrupt the balance and structure of the service architecture.

To alleviate the problems associated with this approach, the Base architecture employs the principle of *receptive execution environments*—systems which can be dynamically configured to host a component of the service software. A collection of receptive execution environments can be constructed either from a set of homogeneous workstations or more diverse resources as required by the service. The distinguishing feature of a receptive execution environment, however, is that the service is "grown" on top of a fertile platform; functionality is *pushed into* each node as appropriate for the application. Each node in the Base can be remotely and dynamically configured by uploading service code components as needed, allowing us to delay the decision about the details of a particular node's specialization as far as possible into the service construction and maintenance lifecycle.[2]

As we will see in section 3, our approach has been to make a single assumption of homogeneity across systems in a Base: a Java Virtual Machine is available on each node. In doing so, we raise the bar of service construction by providing a common instruction set across all nodes, unified views on threading models, underlying system APIs (such as socket and filesystem access), as well as the usual strong typing and safety features afforded by the Java environment. Because of these provisions, any service component can be pushed into any node in our Base and be expected to execute, subject to local resource considerations (such as whether a particular node has access to a disk array or a CD drive). Assuming that every node is capable of receiving Java bytecodes, however, means that techniques generally applied to mobile code systems [21, 13, 28, 32, 18] can be employed internally to the Base: the administrator can deploy service components by uploading Java classes into nodes as needed, and the service can push itself towards resources redistributing code amongst the participating nodes. Furthermore, because in this environment we are restricting our use of code mobility to deploying local code within the scope of a single, trusted administrative domain, some of the security difficulties of mobile code are reduced.

[2] We rely on two mechanisms for mobile code security:we restrict the use of mobile code inside the Base to code that originates from trusted sources within the Base itself, and we use the Java Security Manager mechanism to sandbox this mobile code. Our research goals, however, do not include solving the mobile code security problem.

2.3 Dynamic Redirector Stub Generation

One challenge for clustered servers is to present a single service interface to the outside world, and to mask load-balancing and failover mechanisms in the cluster. The naive solution is to have a single front-end machine that clients first contact; the front-end then dispatches these incoming requests to one of several back-end machines. Failures can be hidden through the selection of another back-end machine, and load-balancing can be directly controlled by the front-end's dispatch algorithm. Unfortunately, the front-end can become a performance bottleneck and a single point of failure [11, 8]. One solution to these problems is to use multiple front-end machines, but this introduces new problems of naming (how clients determine which front-end to use) and consistency (whether the front-ends mutually agree on the back-end state of the cluster). The naming problem can be addressed in a number of ways, such as round-robin DNS [6], static assignment of front-ends to clients, or "lightweight" redirection in the style of scalable Web servers [17]. The consistency problem can be solved through one of many distributed systems techniques [4] or ignored if consistent state is unimportant to the front-end nodes.

The Base architecture takes another approach to cluster access indirection: the use of *dynamically-generated Redirector Stubs*. A stub is client-side code which provides access to a service; a common example is the stub code generated for CORBA/IIOP [25] and Java Remote Method Invocation (RMI) [23] systems. The stub code runs on the client and converts client requests for service functionality (such as Java method calls) into network messages, marshalling request parameters and unmarshalling results. In the case of Java RMI, clients download stubs on-demand from the server.

Base services employ a similar technique to RPC stub generation except that the Redirector Stub for a service is dynamically generated at run-time and contains embedded logic to select from a set of nodes within the cluster (figure 2). Load balancing is implemented within this "Redirector Stub", and failover is accomplished by reissuing failed or timed-out service calls to an alternative back-end machine. The redirection logic and information about the state of the Base is built up by the Base and advertised to clients periodically; clients obtain the Redirector Stubs from a registry. This methodology has a number of significant implications about the nature of services, namely that they must be idempotent and maintain self-consistency across a

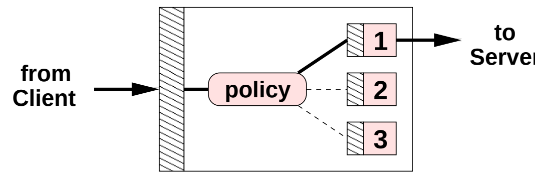

Figure 2: **A "Redirector Stub":** embedded inside a Redirectory Stub are several RPC stubs with the same interface, each of which communicates with a different service instance inside a Base.

service's instances on different nodes in the Base. In section 3.3.1, we will discuss an implementation of a cluster-wide distributed data structure that simplifies the task of satisfying these implications.

Client applications can be coded *without knowledge of the Redirector Stub logic*; by moving failover and load-balancing functionality to the client, the use of front-end machines can be avoided altogether. This is similar to the notion of smart clients [34], but with the intelligence being injected into the client at run-time instead of being compiled in.

3 Implementation

Our prototype Base implementation (written in Java) is called the *MultiSpace*. It serves to demonstrate the effectiveness of our architecture in terms of facilitating the construction of flexible services, and to allow us to begin explorations into the issues of our platform's scalability. The MultiSpace implementation has three layers: the bottom layer is a set of communications primitives (NinjaRMI); the middle layer is a single-node execution environment (the iSpace); and the top layer is a set of multiple node abstractions (the MultiSpace layer). We describe each of these in turn, from the bottom up.

3.1 NinjaRMI

A rich set of high performance communications primitives is a necessary component of any clustered environment. We chose to make heavy use of Java's Remote Method Invocation (RMI) facilities for performing RPC-like [5] calls across nodes in the cluster, and between clients and services. When a caller invokes an RMI method, stub code intercepts the invocation, marshalls arguments, and sends them to a remote "skeleton" method handler for unmarshalling and execution. Using RMI as the finest granularity communication data unit in our clustered environment has many useful properties. Because method invocations have completely encapsulated, atomic semantics, retransmissions or communication failures are easy to reason about—they correspond to either successful or failed method invocations, rather than partial data transmissions.

However, from the point of view of clients, if a remote method invocation does not successfully return, it can be impossible for the client to know whether or not the method was successfully invoked on the server. The client has two choices: it can reinvoke the method call (and risk calling the same method twice), or it can assume that the method was not invoked, risking that the method was in fact invoked, successfully or unsuccessfully with an exception, but results were not returned to the client. Currently, on failure our Redirector Stubs will retry using a different, randomly chosen service stub; in the case of many successive failures, the Redirector Stub will return an exception to the caller. It is because of the at-least-once semantics implied by these client-side reinvocations that we must require services to be idempotent. The exploration of different retry policies inside the Redirector Stubs is an area of future research.

3.1.1 NinjaRMI enhancements to Sun's RMI

NinjaRMI is a ground-up reimplementation of Sun's Java Remote Method Invocation for use by components within the Ninja system. NinjaRMI was designed to permit maximum flexibility in implementation options. NinjaRMI provides three interesting transport-level RMI enhancements. First, it provides a unicast, UDP-based RMI that allows clients to call methods with "best-effort" semantics. If the UDP packet containing the marshalled arguments successfully arrives at the service, the method is invoked; if not, no retransmissions are attempted. Because of this, we enforce the requirement that such methods do not have any return values. This transport is useful for beacons, log entries, or other such side-effect oriented uses that do not require reliability. Our second enhancement is a multicast version of this unreliable transport. RMI services can associate themselves with a multicast group, and RMI calls into that multicast group result in method invocations on all listening services. Our third enhancement is to provide very flexible certificate-based authentication and encryption support for reliable, unicast RMI. Endpoints in an RMI session can associate themselves with digital certificates issued by a certification authority. When the TCP-connection underlying an RMI session is established, these certificates are exchanged by the RMI layer and verified at each endpoint. If the certificate verification succeeds, the remainder of the communication over that TCP connection is encrypted using a Triple DES session key obtained from a Diffie-Hellman key exchange. These security enhancements are described in [12].

Packaged along with our NinjaRMI implementation is an interface compiler which, when given an object that exports an RMI interface, generates the client-side stub and server-side skeleton stubs for that object. All source code for stubs and skeletons can be generated dynamically at run-time, allowing the Ninja system to leverage the use of an intelligent code-generation step when constructing wrappers for service components. We use this to introduce the level of indirection needed to implement Redirector Stubs—stubs can be overloaded to cause method invocations to occur on many remote nodes, or for method invocations to fail over to auxiliary nodes in the case of a primary node's failure.

3.1.2 Measurements of NinjaRMI

Method	**Local** method	**Sun RMI**	**Ninja RMI**
`f(void)`	0.19 μs	0.83 ms	0.82 ms
`f(int)`	0.20 μs	0.84 ms	0.85 ms
`int f(int)`	0.18 μs	0.85 ms	0.84 ms
`int f( int,int,int,int)`	0.22 μs	0.88 ms	0.86 ms
`f(byte[100])`	0.19 μs	1.06 ms	1.05 ms
`f(byte[1000])`	0.20 μs	1.20 ms	1.09 ms
`f(byte[10000])`	0.21 μs	2.21 ms	2.23 ms
`byte[100] f(int)`	0.19 μs	1.07 ms	1.00 ms
`byte[1000] f(int)`	0.20 μs	1.17 ms	1.01 ms
`byte[10000] f(int)`	0.20 μs	2.20 ms	2.10 ms

Table 1: **NinjaRMI microbenchmarks:** These benchmarks were gathered on two 400Mhz Pentium II based machines, each with 128MB of physical memory, connected by a switched 100 Mb/s Ethernet, and using Sun's JDK 1.1.6v2 with the TYA just-in-time compiler on Linux 2.0.36. For the sake of comparison, UDP round-trip times between two C programs were measured at 0.185 ms, and between two Java programs at 0.316 ms.

As shown in table 1, NinjaRMI performs as well as or better than Sun's Java RMI package. Given that a null RMI invocation cost 0.82 ms and that a round-trip across the network and through the JVMs cost 0.316 ms, we conclude that the difference (roughly 0.5 ms) is RMI marshalling and protocol overhead. Profiling the code shows that the main contributor to this overhead is object serialization, specifically the use of methods such as `java.io.ObjectInputStream.read()`.

3.2 iSpace

A Base consists of a number of workstations, each running a suitable receptive execution environment for single-node service components. In our prototype, this receptive execution environment is the *iSpace*: a Java Virtual Machine (JVM) that runs a component loading service into which Java classes can be pushed as needed. The iSpace is responsible for managing component resources, naming, protection, and security. The iSpace exports the component loader interface via NinjaRMI; this interface allows a remote client to obtain a list of components running on the iSpace, obtain an RMI stub to a particular component, and upload a new service component or kill a component already running on the iSpace (subject to authentication). Service components running on the iSpace are protected from one another and from the surrounding execution environment in three ways:

1. each component is coded as a Java class which provides protection from hard crashes (such as null pointer dereferences),

2. components are separated into thread groups; this limits the interaction one component can have with threads of another, and

3. all components are subject to the iSpace Security Manager, which traps certain Java API calls and determines whether the component has the credentials to perform the operation in question, such as file or network access.

Other assumptions must be made in order to make this approach viable. In essence, we are relying on the JVM to behave and perform like a miniature operating system, even though it was not designed as such. For example, the Java Virtual Machine does not provide adequate protection between threads of multiple components running within the same JVM: one component could, for example, consume the entire CPU by running indefinitely within a non-blocking thread. Here, we must assume that the JVM employs a preemptive thread scheduler (as is true in the Sun's Solaris Java environment) and that fairness can be guaranteed through its use. Likewise, the iSpace Security Manager must utilize a strategy for resource management which ensures both fairness and safety. In this respect iSpace has similar goals to other systems which provide multiple protection domains within a single JVM, such as the JKernel [16]. However, our approach does not necessitate a re-engineering of the Java runtime libraries, particularly because intra-JVM thread communication is not a high-priority feature.

3.3 MultiSpace

The highest layer in our implementation is the MultiSpace layer, which tethers together a collection of iSpaces (figure 3). A primary function of this layer is to provide each iSpace with a replicated registry of all service instances running in the cluster.

A MultiSpace service inherits from an abstract MultiSpaceService class, whose constructor registers each service instance with a "MultiSpaceLoader" running on the local iSpace. All MultiSpaceLoaders in a cluster cooperate to maintain the replicated registry; each one periodically sends out a multicast beacon[3] that carries its list of local services to all other nodes in the MultiSpace, and also listens for multicast messages from other MultiSpace nodes. Each MultiSpaceLoader builds an independent version of the registry from these beacons. The registries are thus soft-state, similar in nature to the cluster state maintained in [11] and [1]—if an iSpace node goes down and comes back up, its MultiSpaceLoader simply has to listen to the multicast channel to rebuild its state. Registries on different nodes may see temporary periods of inconsistency as services are pushed into the MultiSpace or moved across nodes in the MultiSpace, but in steady state, all nodes asymptotically approach consistency. This consistency model is similar in nature to that of Grapevine [10].

The multicast beacons also carry RMI stubs for each local service component, which implies that any service instance running on any node in the MultiSpace can identify and contact any other service in the MultiSpace. It also means that the MultiSpaceLoader on every node has enough information to construct Redirector Stubs for all of the services in the MultiSpace, and advertise those Redirector Stubs to off-cluster clients through a "service discovery service"[9].[4] Each RMI stub embedded in

[3]Currently, IP multicast is used for this purpose — the multicast channel that beacons are sent over thus defines the logical scope and boundary of an individual MultiSpace. We intend to replace this transport with multicast NinjaRMI.

[4]The service discovery service (or SDS) implementation consists of an XML search engine that allows client programs to locate services based on arbitrary XML predicates.

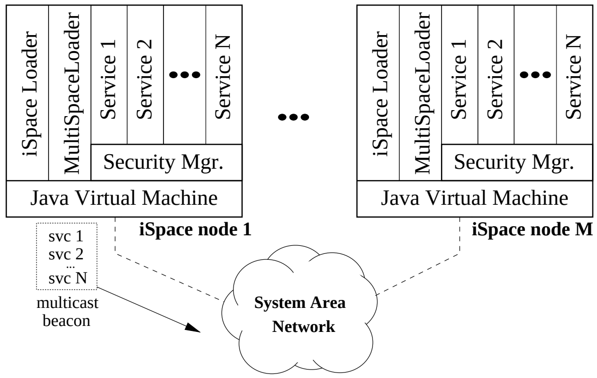

Figure 3: **The MultiSpace implementation:** MultiSpace services are instantiated on top of a sandbox (the Security Manager), and run inside the context of a Java virtual machine (JVM).

a multicast beacon is (based on our observations) roughly 500 bytes in average length; there is therefore an important tradeoff between the beacon frequency (and therefore freshness of information in the MultiSpaceLoaders) and the number of services whose stubs are being beaconed that will ultimately affect the scalability of the MultiSpace.

Service instances can elect to receive multicast beacons from other MultiSpace nodes; a service can use this mechanism to become aware of its peers running elsewhere in the cluster. If a service overrides the standard beacon class, it can augment its beacons with additional information, such as the load that it is currently experiencing. Services that want to call out to other services could thus make coarse grained load balancing decisions without requiring a centralized load manager.

3.3.1 Built-In MultiSpace Services

Included with the MultiSpace implementation are two services that enrich the functionality available to other MultiSpace services: the distributed hash table, and the uptime monitoring service.

Distributed hash table: as mentioned in section 2.3, due to the Redirector Stub mechanism MultiSpace service instances must maintain self-consistency across the nodes of the cluster. To make this task simpler, we have provided service authors with a distributed, replicated, fault-tolerant hash table that is implemented in C for the sake of efficiency. The hash table is designed to present a consistent view of data across all nodes in the cluster, and as such, services may use it to rendezvous in a style similar to Linda [2] or IBM's T-Spaces [33]. The current implementation is moderately fast (it can handle more than 1000 insertions per second of 500 byte entries on a 4 node 100Mb/s MultiSpace cluster), is fault tolerant, and transparently mask multiple node failures. However, it does not yet provide all of the consistency guarantees that some services would ideally prefer, such as on-line recovery of crashed state or transactions across multiple operations. The currently implementation is, however, suitable for many Internet-style services for which this level of consistency is not essential.

Uptime monitoring service: Even with the service beaconing mechanism, it is difficult to detect the failure of individual nodes in the cluster. This is partly because of the lack of a clear failure model in Java: a Java service is just a collection of objects, not necessarily even possessing a thread of execution. A service failure may just imply that a set of objects has entered a mutually inconsistent state. The absence of beacons doesn't necessarily mean that a service instance failure has occurred; the beacons may be lost due to congestion in the internal network, or the beacons may not have been generated because the service instance is overloaded and is busy processing other tasks.

For this reason, we have provided an uptime monitoring abstraction to service authors. If a service running in the MultiSpace implements a well-known Java interface, the infrastructure automatically detects this and begins periodically calling the `doProbe()` method in that interface. By implementing this method, service authors promise to perform an application-level task that demonstrates that the service is accepting and successfully processing requests. By using this application-level uptime check, the infrastructure can explicitly detect when a service instance has failed. Currently, we only log this failure in order to generate uptime statistics, and we rely on the Redirector Stub failover mechanisms to mask these failures.

4 Applications

In this section of the paper, we discuss two applications that demonstrate the validity of our guiding principles and the efficacy of our MultiSpace implementation. The first application, the Ninja Jukebox, abstracts the many independent compact-disc players and local filesystems in the Berkeley Network of Workstations (NOW) cluster into a single pool of available music. The second, Keiretsu, is a three-tiered application that provides instant messaging across heterogeneous devices.

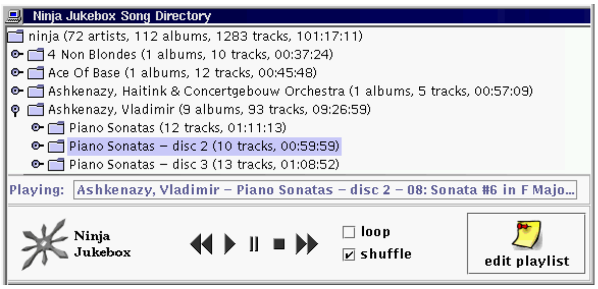

Figure 4: **The Ninja Jukebox GUI:** users are presented with a single Jukebox interface, even though songs in the Jukebox are scattered across multiple workstations, and may be either MP3 files on a local filesystem, or audio CDs in CD-ROM drives.

4.1 The Ninja Jukebox

The original goal of the Ninja jukebox was to harness all of the audio CD players in the Berkeley NOW (a 100+ node cluster of Sun UltraSparc workstations) to provide a single, giant virtual music jukebox to the Berkeley CS graduate students. The most interesting features of the Ninja Jukebox arise from its implementation on top of iSpace: new nodes can be dynamically harnessed by pushing appropriate CD track "ripper" services onto them, and the features of the Ninja Jukebox are simple to evolve and customize, as evidenced by the seamless transformation of the service to the batch conversion of audio CDs to MP3 format, and the authenticated transmission of these MP3s over the network.

The Ninja Jukebox service is decomposed into three components: a master directory, a CD "ripper" and indexer, and a gateway to the online CDDB service [22] that provides artist and track title information given a CD serial number. The ability to push code around the cluster to grow the service proved to be exceptionally useful, since we didn't have to decide a priori which nodes in the cluster would house CDs—we could dynamically push the ripper/indexer component towards the CDs as the CDs were inserted into nodes in the cluster. When a new CD is added to a node in the NOW cluster, the master directory service pushes an instance of the ripper service into the iSpace resident on that node. The ripper scans the CD to determine what music is on it. It then contacts a local instance of the CDDB service to gather detailed information about the CD's artist and track titles; this information is put into a playlist which is periodically sent to the master directory service. The master directory incorporates playlists from all of the rippers running across the cluster into a single, global directory of music, and makes this directory available over both RMI (for song browsing and selection) and HTTP (for simple audio streaming).

After the Ninja Jukebox had been running for a while, our users expressed the desire to add MP3 audio files to the Jukebox. To add this new behavior, we only had to subclass the CD ripper/indexer to recognize MP3 audio files on the local filesystem, and create new Jukebox components that batch converted between audio CDs and MP3 files. Also, to protect the copyright of the music in the system we added access control lists to the MP3 repositories, with the policy that users could only listen to music that they had added to the Jukebox.[5] We then began pushing this new subclass to nodes in the system, and our system evolved while it was running.

The performance of the Ninja Jukebox is completely dominated by the overhead of authentication and the network bandwidth consumed by streaming MP3 files. The first factor (authentication overhead) is currently benchmarked at a crippling 10 seconds per certificate exchange, entirely due to a pure Java implementation of a public key cryptosystem. The second factor (network consumption) is not quite as crippling, but still significant: each MP3 consumes at least 128 Kb/s, and since the MP3 files are streamed over HTTP, each transmission is characterized by a large burst as the MP3 is pushed over the network as quickly as possible. Both limitations can be remedied with significant engineering, but this would be beyond the scope of our research.

4.2 Keiretsu: The Ninja Instant-Messaging Service

Keiretsu[6] is a MultiSpace service that provides instant messaging between heterogeneous devices: Web browsers, one- or two-way pagers, and PDAs such as the Palm Pilot (see Figure 5). Users are able to view a list of other users connected to the Keiretsu service, and can send short text messages to other users. The service component of Keiretsu exploits the MultiSpace features: Keiretsu service instances use the soft-state registry of peer nodes in order to exchange client routing information across the cluster, and automatically generated Redirector Stubs are handed out to clients for use in communicating with Keiretsu nodes.

[5] This ACL policy is enforced using the authentication extensions to NinjaRMI described in 3.1.1.

[6] *Keiretsu* is a Japanese concept in which a group of related companies work together for each other's mutual success.

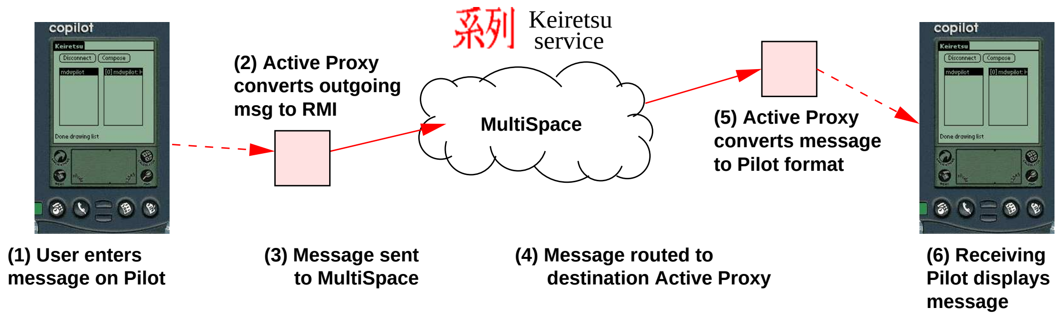

Figure 5: **The Keiretsu Service**

Keiretsu is a three-tired application: simple client devices (such as pagers or Palm Pilots) that cannot run a JVM connect to an Active Proxy, which can be thought of as a simplified iSpace node meant to run soft-state mobile code. The Active Proxy converts simple text messages from devices into NinjaRMI calls into the Keiretsu MultiSpace service. The Active Proxies are assumed to have enough sophistication to run Java-based mobile code (the protocol conversion routines) and speak NinjaRMI, while rudimentary client devices need only speak a simple text-based protocol.

As described in section 2.3, Redirector Stubs are used to access the back-end service components within a MultiSpace by pushing load-balancing and failover logic towards the client—in the case of simple clients, Redirector Stubs execute in the Active Proxy. For each protocol message received by an Active Proxy from a user device (such as "send message M to user U"), the Redirector Stub is invoked to call into the MultiSpace.

Because the Keiretsu proxy is itself a mobile Java component that runs on an iSpace, the Keiretsu proxy service can be pushed into appropriate locations on demand, making it easy to bootstrap such an Active Proxy as needed. State management inside the Active Proxy is much simpler than state management inside a Base—the only state that Active Proxies maintain is the session state for connected clients. This session state is soft-state, and it does not need to be carefully guarded, as it can be regenerated given sufficient intelligence in the Base, or by having users manually recover their sessions.

Rudimentary devices are not the only allowable members of a Keiretsu. More complex clients that can run a JVM speak directly to the Keiretsu, instead of going through an Active Proxy. An example of such a client is our e-mail agent, which attaches itself to the Keiretsu and acts as a gateway, relaying Keiretsu messages to users over Internet e-mail.

4.2.1 The Keiretsu MultiSpace service

```
public void identifySelf(
  String clientName,
  KeiretsuClientIF clientStub);

public void disconnectSelf(String clientName);

public void injectMessage(KeiretsuMessage msg);

public String[] getClientList();
```

Figure 6: **The Keiretsu service API**

The MultiSpace service that performs message routing is surprisingly simple. Figure 6 shows the API exported by the service to clients. Through the `identifySelf` method, a client periodically announces its presence to the Keiretsu, and hands the Keiretsu an RMI stub which the service will use to send it messages. If a client stops calling this method, the Keiretsu assumes the client has disconnected; in this way, participation in the Keiretsu is treated as a lease. Alternately, a client can invalidate its binding immediately by calling the `disconnectSelf` method. Messages are sent by calling the `injectMessage` method, and clients can obtain a list of other connected clients by calling the `getClientList` method.

Inside the Keiretsu, all nodes maintain a soft-state table of other nodes by listening to MultiSpace beacons, as discussed in section 3.3. When a client connects to a Keiretsu node, that node sends the client's RMI stub to all other nodes; all Keiretsu nodes maintain individual tables of these client bindings. This means that in steady state, each node can route messages to any client.

Because clients access the Keiretsu service through Redirector Stubs, and because Keiretsu nodes replicate service state, individual nodes in the

Keiretsu can fail and service will continue uninterrupted, at the cost of capacity and perhaps performance. In an experiment on a 4-node cluster, we demonstrated that the service continued uninterrupted even when 3 of the 4 nodes went down. The Keiretsu source code consists of 5 pages of Java code; however, most of the code deals with managing the soft-state tables of the other Keiretsu nodes in the cluster and the client RMI stub bindings. The actual business of routing messages to clients consists of only *half a page of Java code*—the rest of the service functionality (namely, building and advertising Redirector Stubs, tracking service implementations across the cluster, and load balancing and failover across nodes) is hidden inside the MultiSpace layer. We believe that the MultiSpace implementation is quite successful in shielding service authors from a significant amount of complexity.

4.2.2 Keiretsu Performance

We ran an experiment to measure the performance and scalability of our MultiSpace implementation and the Keiretsu service. We used a cluster of 400 MHz Pentium II machines, each with 128 MB of physical memory, connected by a 100 Mb/s switched Ethernet. We implemented two Keiretsu clients: the "speedometer", which open up a parameterizable number of identities in the Keiretsu and then waits to receive messages, and the "driver", which grabs a parameterizable number of Redirector Stubs to the Keiretsu, downloads a list of clients in the Keiretsu, and then blasts 75 byte messages to randomly selected clients as fast as it can.

We started our Keiretsu service on a single node, and incrementally grew the cluster to 4 nodes, measuring the maximum message throughput obtained for $10x$, $50x$, and $100x$ "speedometer" receivers, where x is the number of nodes in the cluster. To achieve maximum throughput, we added incrementally more "driver" connections until message delivery saturated. The drivers and speedometers were located on many dedicated machines, connected to the Keiretsu cluster by the same 100 Mb/s switched Ethernet. Table 2 shows our results.

For a small number of receivers (10 per node), we observed linear scaling in the message throughput. This is because each node in the Keiretsu is essentially independent: only a small amount of state is shared (the client stubs for the 10 receivers per node). In this case, the CPU was the bottleneck, likely due to Java overhead in message processing and argument marshalling and unmarshalling.

For larger number of receivers, we observed a breakdown in scaling when the total number of receivers reached roughly 200 (i.e. 3-4 nodes at 50 receivers per node, or 2 nodes at 100 receivers per node). The CPU was still the bottleneck in these cases, but most of the CPU time was spent processing the client stubs exchanged between Keiretsu nodes, rather than processing the clients' messages. This is due to poor design of the Keiretsu service; we did not need to exhange client stubs as frequently as we did, and we should have simply exchanged timestamps for previously distributed stubs rather than repeatedly sending the same stub. This limitation could be also removed by modifying the Keiretsu service to inject client stubs in a distributed hash table, and rely on service instances to pull the stubs out of the table as needed. However, a similar N^2 state exchange happens at the MultiSpace layer with the multicast exchange of service instance stubs; this could potentially become another scaling bottleneck for large clusters.

# Nodes	# Clients per node	Max. message throughput (msgs / s)
1	10	246 ± 4
	50	200 ± 8
	100	195 ± 10
2	10	420 ± 10
	50	300 ± 20
	100	260 ± 20
3	10	490 ± 15
	50	370 ± 20
	100	160 ± 15
4	10	570 ± 15
	50	210 ± 10
	100	120 ± 10

Table 2: **Keiretsu performance:** These benchmarks were run on 400Mhz Pentium II machines, each with 128MB of physical memory, connected by a 100 Mb/s switched Ethernet, using Sun's JDK 1.1.6v2 with the TYA just-in-time compiler on Linux 2.2.1, and sending 75 byte Keiretsu messages.

5 Discussion and Future Work

Our MultiSpace and service implementation efforts have given some insights into our original design principles, and into the use of Java as an Internet service construction language. In this section of the paper, we delve into some of these insights.

5.1 Code Mobility as a Service Construction Primitive

When we were designing the MultiSpace, we knew that code mobility would be a powerful tool. We originally intended to use code mobility for delivering code to clients (which we do in the form of Redirector Stubs), and for it to be used by clients to upload customization code into the MultiSpace (which has not been implemented). However, code mobility turned out to be useful *inside* the MultiSpace as a mechanisms for structuring services, and distributing service components across the cluster.

Code mobility solved a software distribution problem in the Jukebox, without us realizing that software distribution might become a problem. When we updated the ripper service, we needed to distribute the new functionality to all of the nodes in the cluster that would potentially have CD's inserted into them. Code mobility also partially solved the service location problem in the Jukebox: the ripper services depend on the CDDB service to gather detailed track and album information, but the ripper has no easy way to know where in the cluster the CDDB service is running. Using code mobility to push the CDDB service onto the same node as the ripper, we enforced the invariant that the CDDB service is colocated with the ripper.

5.2 Bases are a Simplifying Principle, but not a Complete Solution

The principle of solving complex service problems in a Base makes it easier to reason about the interactions between services and clients, and to ensure that difficult tasks like state management are dealt with in an environment in which there is a chance of success. However, this organizational principle alone is not enough to solve the problem of constructing highly available services. Given the controlled environment of a Base, service authors must still construct services to ensure consistency and availability. We believe that a Base can provide primitives that further simply service authors' jobs; the Redirector Stub is an example of such a primitive.

While building the Ninja Jukebox and Keiretsu, we made observations about how we achieved availability and consistency. Most of the code in these services dealt with distributing and maintaining tables of shared state. In the Ninja Jukebox, this state was the list of available music. In Keiretsu, this state was the list of other Keiretsu nodes, and the tables of client stub bindings. The distributed hash table was not yet complete when these two services were being implemented. If we had relied on it instead of our ad-hoc peer state exchange mechanisms, much of the services' source code would have been eliminated.

In both of these services, work queues are not explicitly exposed to service authors. These queues are hidden inside the thread scheduler, since NinjaRMI spawns a thread per connected client. This design decision had repercussions on service structure: each service had to be written to handle multithreading, since service authors must handle consistency within a single service instance as well as across instances throughout the cluster. Providing a mechanism to expose work queues to service authors may simplify the structure of some services, for example if services serialize requests to avoid issues associated with multithreading.

In the current MultiSpace, service instances must explicitly keep track of their counterparts on other nodes and spawn new services when load or availability demands it. A useful primitive would be to allow authors to specify conditions that dictate when service instances are spawned or pushed to a specific node, and to allow the MultiSpace infrastructure to handle the triggering of these conditions.

5.3 Java as an Internet-service construction environment

Java has proven to be an invaluable tool in the development of the Ninja infrastructure. The ability to rapidly deploy cross-platform code components simply by assuming the existence of a Java Virtual Machine made it easy to construct complex distributed services without concerning oneself with the heterogeneity of the systems involved. The use of RMI as a strongly-typed RPC, tied very closely to the Java language semantics, makes distributed programming comparably simple to single node development. The protection, modularization, and safety guarantees provided by the Java runtime environment make dynamic dissemination of code components a natural activity. Similarly, the use of Java class reflection to generate new code wrappers for existing components (as with Redirector Stubs) provides automatic indirection at the object level.

Java has a number of drawbacks in its current form, however. Performance is always an issue, and work on just-in-time [14, 24] and ahead-of-time [27] compilation is addressing many of these problems. The widely-used JVM from Sun Microsystems exhibits a large memory footprint (we have observed 3-4 MB for "Hello World", and up to 30

MB for a relatively simple application that performs many of memory allocations and deallocations[7]), and crossing the boundary from Java to native code remains an expensive operation. In addition, the Java threading model permits threads to be non-preemptive, which has serious implications for components which must run in a protected environment. Our approach has been to use only Java Virtual Machines which employ preemptive threads.

We have started an effort to improve the performance of the Java runtime environment. Our initial prototype, called Jaguar, permits direct Java access to hardware resources through the use of a modified just-in-time compiler. Rather than going through the relatively expensive Java Native Interface for access to devices, Jaguar generates machine code for direct hardware access which is inlined with compiled Java bytecodes. We have implemented a Jaguar interface to a VIA[8] enabled fast system area network, obtaining performance equivalent to VIA access from C (80 microseconds round-trip time for small messages and over 400 megabits/second peak bandwidth). We believe that this approach is a viable way to tailor the Java environment for high-performance use in a clustered environment.

6 Related Work

Directly related to the Base architecture is the TACC [11] platform, which provides a cluster-based environment for scalable Internet services. In TACC, service components (or "workers") can be written in a number of languages and are controlled by a front-end machine which dispatches incoming requests to back-end cluster machines, incorporating load-balancing and restart in the case of node failure. TACC workers may be chained across the cluster for composable tasks. TACC was designed to support Internet services which perform data transformation and aggregation tasks. Base services can additionally implement long-lived and persistent services; the result is that the Ninja approach addresses a wider set of potential applications and system-support issues. Furthermore, Base services can dynamically created and destroyed through the iSpace loader interface on each MultiSpace node—TACC did not have this functionality.

[7]There are no explicit deallocations in Java — by "deallocation", we mean discarding all references to an object, thus enabling it to be garbage collected.

[8]VIA is the Virtual Interface Architecture [26], which specifies an industry-standard architecture for high-bandwidth, low-latency communication within clusters.

Sun's JINI [31] architecture is similar to the the Base architecture in that it proposes to develop a Java-based *lingua franca* for binding users, devices, and services together in an intelligent, programmable Internet-wide infrastructure. JINI's use of RMI as the basic communication substrate and use of code mobility for distributing service and device interfaces has a great deal of rapport with our approach. However, we believe that we are addressing problems which JINI does not directly solve: providing a hardware and software environment supporting scalable, fault-tolerant services is not within the JINI problem domain, nor is the use of dynamically-generated code components to act as interfaces into services. However, JINI has touched on issues such as service discovery and naming which have not yet been dealt with by the Base architecture; likewise, JINI's use of JavaSpaces and "leases" as a persistent storage model may interact well with the Base service model.

ANTS [30, 29] is a system that enables the dynamic deployment of mobile code which implements network protocols within Active Routers. Coded in Java, and utilizing techniques similar to those in the iSpace environment, ANTS has a similar set of goals in mind as the Base architecture. However, ANTS uses mobile code for processing each packet passing through a router; Base service components are executed on the granularity of an RMI call. Liquid Software [19] and Joust [15] are similar to ANTS in that they propose an environment which uses mobile code to customize nodes in the network for communications oriented tasks. These systems focused on adapting systems at the level of network protocol code, while the Base architecture uses code mobility for distribution of service components both internally to and externally from a Base.

SanFrancisco [7] is a platform for building distributed, object-oriented business applications. Its primary goal is to simplify the development of these applications by providing developers a set of industrial-strength "Foundation" objects which implement common functionality. As such, SanFrancisco is very similar to Sun's Enterprise Java Beans in that it provides a framework for constructing applications using reusable components, with SanFrancisco providing a number of generic components to start with. MultiSpace addresses a different set of goals than Enterprise Java Beans and SanFrancisco in that it defines a flexible runtime environment for services, and MultiSpace intends to provide scalability and fault-tolerance by leveraging the flexibility of a component architecture. MultiSpace services could be built using the EJB or SanFrancisco model (extended to expose the MultiSpace functionality), but these issues appear to be orthogonal.

The Distributed Computing Environment (DCE) [20] is a software suite that provides a middleware platform that operates on many operating systems and environments. DCE abstracts away many OS and network services (such as threads, security, a directory service, and RPC) and therefore allows programmers to implement DCE middleware independent of the vagaries of particular operating systems. DCE is rich, robust, but notoriously heavyweight, and its focus is on providing interoperable, wide-area middleware. MultiSpace is far less mature, but focuses instead on providing a platform for rapidly adaptable services that are housed within a single administrative domain (the Base).

7 Conclusions

In this paper, we presented an architecture for a *Base*, a clustered hardware and software platform for building and executing flexible and adaptable infrastructure services. The Base architecture was designed to adhere to three organizing principles: **(1)** solve the challenging service availability and scalability problems in carefully *controlled environments*, allowing service authors to make many assumptions that would not otherwise be valid in an uncontrolled or wide area environment, **(2)** gain service flexibility by decomposing Bases into a number of *receptive execution environments*; and **(3)** introduce a level of indirection between the clients and services through the use of *dynamic code generation techniques*.

We built a prototype implementation (MultiSpace) of the Base architecture using Java as a foundation. This implementation took advantage of the code mobility and dynamic compilation techniques to help in the structuring and deployment of services inside the cluster. The MultiSpace abstracts away load balancing, failover, and service instance detection and naming from service authors. Using the MultiSpace platform, we implemented two novel services: the Ninja Jukebox, and Keiretsu. The Ninja Jukebox implementation demonstrated that code mobility is valuable inside a cluster environment, as it permits rapid evolution of services, and run-time binding of service components to available resources in the cluster. The Keiretsu application demonstrated that our MultiSpace layer successfully reduced the complexity of building new services: the core Keiretsu service functionality was implemented in less than a page of code, but the application was demonstrably fault-tolerant. We also demonstrated that Keiretsu code limited the scalability of this service, rather than any inherent limitation in the MultiSpace layer, although we hypothesized that our use of multicast beacons would ultimately limit the scalability of the current MultiSpace implementation.

8 Acknowledgements

The authors thank Ian Goldberg and David Wagner for their contributions to iSpace, NinjaRMI, and the Jukebox application. We would also like to thank all of the graduate and undergraduate students in the Ninja project for agreeing to be guinea pigs for our platform, and Brian Shiratsuki and Keith Sklower for their help in the system of the MultiSpace machines and networks. Finally, we would like to thank our anonymous reviewers and our shepherd, Charles Antonelli, for their insight and suggestions for improvement.

References

[1] Elan Amir, Steven McCanne, and Randy Katz. An Active Service Framework and its Application to Real-Time Multimedia Transcoding. In *Proceedings of ACM SIGCOMM '98*, volume 28, pages 178–189, October 1998.

[2] B. Anderson and D. Shasha. Persistent Linda: Linda + Transactions + Query Processing. In *Springer-Verlag Lecture Notes in Computer Science 574*, Mont-Saint-Michel, France, June 1991.

[3] Thomas E. Anderson, David E. Culler, and David Patterson. A Case for NOW (Networks of Workstations). *IEEE Micro*, 12(1):54–64, February 1995.

[4] Ken Birman, Andre Schiper, and Pat Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, 1991.

[5] Andrew D. Birrell and Bruce Jay Nelson. Implementing Remote Procedure Call. *ACM Transactions on Computing Systems*, 2(1):39–59, February 1984.

[6] T. Brisco. RFC 1764: DNS Support for Load Balancing, April 1995.

[7] IBM Corporation. IBM SanFrancisco product homepage. `http://www.software.ibm.com/ad/sanfrancisco/`.

[8] Inktomi Corporation. The Technology Behind HotBot. `http://www.inktomi.com/whitepap.html`, May 1996.

[9] Steven Czerwinski, Ben Y. Zhao, Todd Hodes, Anthony Joseph, and Randy Katz. An Architecture

for a Secure Service Discovery Service. In *Proceedings of MobiCom '99*, Seattle, WA, August 1999. ACM.

[10] A.D. Birrell et al. Grapevine: An Exercise in Distributed Computing. *Communications of the Association for Computing Machinery*, 25(4):3–23, Feb 1984.

[11] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-Based Scalable Network Services. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, St.-Malo, France, October 1997.

[12] Ian Goldberg, Steven D. Gribble, David Wagner, and Eric A. Brewer. The Ninja Jukebox. In *Submitted to the 2nd USENIX Symposium on Internet Technologies and Systems*, Boulder, Colorado, USA, October 1999.

[13] Robert S. Gray. Agent Tcl: A Flexible and Secure Mobile-Agent System. In *Proceedings of the Fourth Annual Usenix Tcl/Tk Workshop*. USENIX Association, 1996.

[14] The Open Group. The Fajita Compiler Project. `http://www.gr.opengroup.org/java/compiler/fajita/index-b.htm`, 1998.

[15] J. Hartman, L. Peterson, A. Bavier, P. Bigot, P. Bridges, B. Montz, R. Piltz, T. Proebsting, and O. Spatscheck. Joust: A Platform for Liquid Software. In *IEEE Network (Special Edition on Active and Programmable Networks)*, July 1998.

[16] Chris Hawblitzel, Chi-Chao Chang, Grzegorz Czajkowski, Deyu Hu, and Thorsten von Eicken. Implementing Multiple Protection Domains in Java. In *Proceedings of the 1998 Usenix Annual Technical Conference*, June 1998.

[17] Open Group Research Institute. Scalable Highly Available Web Server Project (SHAWS). `http://www.osf.org/RI/PubProjPgs/SFTWWW.htm`.

[18] Dag Johansen, Robbert van Renesse, , and Fred R. Schneider. Operating System Support for Mobile Agents. In *Proceedings of the 5th IEEE Workshop on Hot Topics in Operating Systems*, 1995.

[19] John Hartman and Udi Manber and Larry Peterson and Todd Proebsting. Liquid Software: A New Paradigm for Networked Systems. Technical report, Department of Computer Science, University of Arizona, June 1996.

[20] Brad Curtis Johnson. A Distributed Computing Environment Framework: an OSF Perspective. Technical Report DEV-DCE-TP6-1, the Open Group, June 1991.

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

[22] Ti Kan and Steve Scherf. CDDB Specificaton. `http://www.cddb.com/ftp/cddb-docs/cddb_howto.gz`.

[23] Sun Microsystems. Java Remote Method Invocation—Distributed Computing for Java. `http://java.sun.com/`.

[24] Sun Microsystems. The Solaris JIT Compiler. `http://www.sun.com/solaris/jit`, 1998.

[25] The Object Management Group (OMG). The Common Object Request Broker: Architecture and Specification, February 1998. `http://www.omg.org/library/c2indx.html`.

[26] Virtual Interface Architecture Organization. Virtual Interface Architecture Specification version 1.0, December 1997. `http://www.viarch.org`.

[27] Todd A. Proebsting, Gregg Townsend, Patrick Bridges, John H. Hartman, Tim Newsham, and Scott A. Watterson. Toba: Java for Applications—A Way Ahead of Time (WAT) Compiler. In *Proceedings of the Third USENIX Conference on Object-Oriented Technologies (COOTS)*, Portland, Oregon, USA, June 1997.

[28] Joseph Tardo and Luis Valente. Mobile Agent Security and Telescript. In *Proceedings of the 41st International Conference of the IEEE Computer Society (CompCon '96)*, February 1996.

[29] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden. A Survey of Active Network Research. Active Networks home page (MIT Telemedia, Networks and Systems group), 1996.

[30] David L. Tennenhouse and David J. Wetherall. Towards an Active Network Architecture. In *ACM SIGCOMM '96 (Computer Communications Review)*. ACM, 1996.

[31] Jim Waldo. Jini Architecture Overview. Available at `http://java.sun.com/products/jini/whitepapers`.

[32] James E. White. Telescript Technology: The Foundation for the Electronic Marketplace, 1991. `http://www.generalmagic.com`.

[33] P. Wyckoff, S. W. McLaughry, T. J. Lehman, and D. A. Ford. TSpaces. *IBM Systems Journal*, 37(3), April 1998.

[34] C. Yoshikawa, B. Chun, P. Eastham, A. Vahdat, T. Anderson, and D. Culler. Using Smart Clients to Build Scalable Services. In *Proceedings of the Winter 1997 USENIX Technical Conference*, January 1997.