The Ninja Architecture for Robust Internet-Scale Systems and Services

Steven D. Gribble, Matt Welsh, Rob von Behren, Eric A. Brewer, David Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R.H. Katz, Z.M. Mao, S. Ross, B. Zhao
The University of California at Berkeley
http://ninja.cs.berkeley.edu

Abstract

The Ninja project seeks to enable the broad innovation of robust, scalable, distributed Internet services, and to permit the emerging class of extremely heterogeneous devices to seamlessly access these services. Our architecture consists of four basic elements: bases, which are powerful workstation cluster environments with a software platform that simplifies scalable service construction; units, which are the devices by which users access the services; active proxies, which are transformational elements that are used for unit- or service-specific adaptation; and paths, which are an abstraction through which units, services, and active proxies are composed.

Keywords: distributed systems, scalable services, pervasive computing, thin clients, Ninja architecture

1 Introduction

The emerging Internet landscape is populated by rich services of immense scale that are offered to a diverse spectrum of clients. This presents exciting opportunities for innovation in the kinds of services that can be created, but also presents tremendous design and engineering challenges. The traditional suite of information stores, commerce sites, network services, and search engines are being combined in novel ways to provide new services that aggregate and transform many sources of information. In addition, services are presenting themselves in a multitude of forms to match the particular capabilities of PCs, PDAs, webphones, and other devices; this adaptation to diversity raises new notions of service composition and content transformation. Moreover, these new services may be utilized by millions of users.

In this opportunity for innovation and vast delivery lies a deep engineering challenge: a successful service may need to scale to huge levels of load over a short period and it must be continuously available. The Ninja project seeks to address these two goals — enabling broad innovation of service design and easily constructing scalable, robust services — through a distributed service architecture that deals with huge throughput demands and availability requirements in a generic fashion, while facilitating service composition.

The distributed service architecture tackles the problem of ease of authoring scalable, robust services at several levels. At the network architecture level, structure is imposed on the Internet by a partitioning into three tiers (scalable service platforms, transformational intermediaries between devices and services, and the devices themselves) to facilitate state management and consistency while operating in the presence of failures. Deep processing power and persistent storage is provided within the service platform through the use of well-engineered clusters on fast, dedicated networks, while soft state and functional transformations are provided close to the devices. A service is rendered along a path crossing all the tiers. These paths are the natural unit of adaptation, optimization and management.

At the language level, services are written in a type-safe language (Java) to reduce errors and to facilitate composition at well defined interfaces. Code mobility is harnessed to dynamically upload services into the platform. At the system level, a platform provides a set of interfaces and dictates a programming discipline that yields efficiently pipelined services that are robust to excessive load, are replicated to achieve high absolute throughput, and are tolerant of node failures. Services describe themselves to a service discovery service, which itself must scale, so that they can be composed programmatically to yield new services. It is the structure and careful design of the overall platform that simplifies the task of authoring services, because they inherit the approach to scalability, availability, fault-tolerance, data consistency, and persistence from the platform.

We begin in Section 2 with an overview of the entire Ninja platform architecture and an introduction of its basic terms and concepts. Section 3 develops the core service platform, called a base, including the programming model for services, the execution vehicle, and the approach to scalable, persistent state. Section 4 describes the characteristics of emerging devices, called units, which fundamentally rely upon the infrastructure. Section 5 describes the role and function of the transformational intermediaries, called Active Proxies. Section 6 lays the foundation for service composition, showing how services describe themselves and locate other services in the wide area. Section 7 illustrates how services are composed across the platform tiers through the path concept. Section 8 puts these concepts together in four distinct services. The remaining sections discuss related work and future directions.

This work is supported, in part, by the Defense Advanced Research Project Agency (grant DABT 63-98-C-0038) and the National Science Foundation (grant RI EIA-9802069). Support is provided as well by Intel Corporation, Ericsson, Philips, Sun Microsystems, IBM, Nortel Networks, and Compaq.

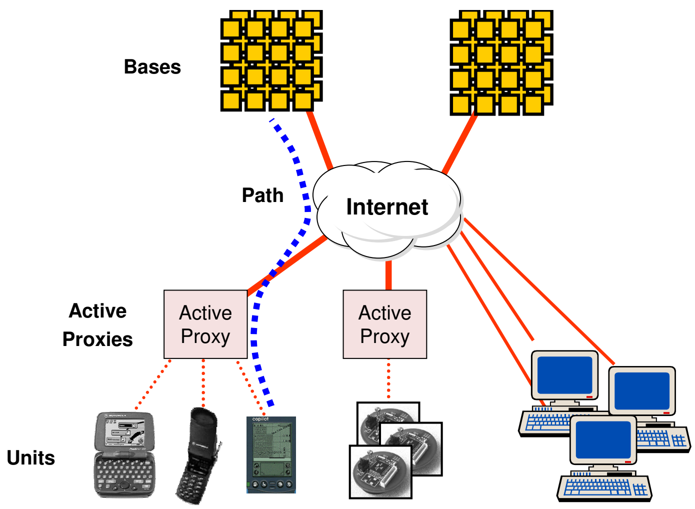

Figure 1: **The Ninja Platform Architecture:** The architecture consists of bases (services running on clusters of workstations), active proxies (stateless or soft-state intermediaries between units and services), units (heterogeneous devices and sensors), and paths (a composition chain across units, proxies, and services in bases).

2 Overview of the Ninja Architecture

In Figure 1, we provide a high-level illustration of the Ninja architecture, decomposing it into four basic elements: **bases**, which are the rich environments that are engineered to support scalable and robust services, **units**, which are the numerous, heterogeneous devices that we wish to support, **active proxies**, which are transformational elements used for device- or service-specific adaptation, and **paths**, an abstraction used to compose the other elements. We motivate and explore each of these in turn.

2.1 Building Robust Services in Bases

We define a **service** as software embedded in the Internet infrastructure that exports a network-accessible, typed, programmatic interface, and that provides strong operational guarantees (such as high availability). The task of building and maintaining services is extremely challenging, since if they are to be depended upon, they must have the essential properties of scalability, availability, fault-tolerance, and data consistency and persistence, all in the face of voluminuous and potentially growing traffic demands. Unfortunately, there is currently a lack of suitable reusable building blocks and design methodologies for service construction.

We address this challenge in part by constraining the execution environment of services: we mandate that the core of the service must run in a well-engineered cluster of workstations, which we call a **base**. Clusters [3] are a natural platform for building Internet services: each cluster node represents an independent failure boundary, which means that replication of computation and data can be used to provide fault-tolerance. A cluster permits incremental scalability, as nodes can be added to increase capacity. Coupled with high performance system area networks, clusters can deliver excellent performance for relatively low cost. Modern cluster networks can achieve greater than 1 Gb/s throughput with 10 to 100 μs latency.

Designing software to run on clusters of workstations is known to be difficult [15]. To simplify the task of authoring new services, we have constructed a cluster-based software platform (called **vSpace**) that allows service authors to concentrate on application-specific functionality, rather than on details related to scalability, fault-tolerance, and composability. Services authored to run on the vSpace platform inherit the essential service properties from the platform, greatly reducing the size and complexity of service code.

vSpace supports the dynamic uploading of new services by trusted or untrusted third parties; we believe this open infrastructure is an important property necessary to sustain the distributed innovation that has lead to the current success of the Internet. Authors can construct their services locally, but then upload their services into bases that are externally maintained.

2.2 Device Diversity

The spectrum of network-attached client devices is growing in diversity and scale. In addition to PCs, laptops, and the now familiar class of PDAs and mobile devices, networks of even more resource-constrained tiny devices such as sensors and actuators are being attached to the Internet. This large family of client devices, which we call **units**, may have limited connectivity and low or intermittent bandwidth, poor computational abilities, and may be able to handle only a small set of data formats and network protocols. We believe that there will be a very large number of units attached to the network, reaching scales of hundreds of millions, and eventually, billions of devices.

Units, by nature, are typically not useful without supporting infrastructure. We assume that units can be easily lost or broken, implying that any state that they manage must be replicated in a durable environment, such as a service running in a base, which can provide vast amounts of highly available, durable storage. Inexpensive or small units may not have enough computational ability to handle the rich set of data types and the growing set of protocols deployed in the Internet, implying that such units must rely on surrogates that adapt content and protocols on their behalf.

Because some units are mobile, they may experience regular periods of disconnected operation. Bases can assist such weakly connected units with the consistency management of data shared across units. Similarly, while a unit is disconnected, a service running on a base can act as the unit's surrogate by responding to requests based on the most recent information in the service's persistent data store.

2.3 Adaptation

The growing number of devices with Internet access capabilities presents a unique set of problems to the designers of Internet-based services. As the demand for continuous access to content is increasing, access to services is being demanded in new environments such as automobiles [26] and kiosks in airplane seats [25], and through new devices such as web-phones. Constructing a service that can be easily and securely used from this diversity of contexts and devices is daunting, because of the huge variation in computational power, network connectivity, and interface capabilities of the devices. Additionally, the weak computational ability of small devices such as pagers and PDAs prevents them from using cryptographic protocols such as SSL to access secure services. In today's Internet, this diversity in client capabilities simply means that most services are inaccessible to clients other than standard home PCs or office workstations.

Rather than forcing services to adapt their content and access protocols to the abilities of all current and future devices, we place transformational intermediaries, called **active proxies**, between devices and services to shield them from each other. An active proxy can transform data types through a process called distillation, adapt protocols (e.g., by converting an SSL connection into a less expensive shared-key encrypted channel for CPU constrained devices), or even adapt the value of content by removing sensitive information before content is displayed on an untrusted access point. Examples of active proxies include wireless basestations, network gateways, firewalls, caching proxies, and transformational proxies. Devices may migrate to a new geographic or administrative domain, and in the process may need to begin using a new active proxy.

2.4 The Composition of Services

Instead of constructing a set of isolated, vertical services that can handle a fixed set of devices, our architecture supports the dynamic composition of horizontal services into a **path**, as well as adaptation along that path. A path is a flow of typed data through multiple services across the wide area, including the interposition of transformational operators to adapt the data into the form expected by the next service or device along the path. An essential feature of services that enables path composition is programmatic access; services must export typed, programmatically-accessible interfaces, as opposed to the untyped, unstructured user interfaces common to services today.

Because paths can be established dynamically, the path creation infrastructure can perform dataflow optimization by examining many different potential paths before deciding on a particular one to use. During the course of this examination, it can weigh the costs of the various paths, and choose a path that optimizes for quality of service, resource consumption, or some other metric. By allowing the optimization process to continue through the lifetime of a given path, the infrastructure adapts the path to the changing characteristics of the execution environment. For example, if a network link becomes overloaded while data is flowing through the path, this flow may be redirected through a different channel to improve the quality of service.

A necessary step to forming a path is being able to locate services to place in that path. The Ninja architecture includes a **service discovery service (SDS)** that allows both human users and programs to locate appropriate services across the wide area based on service attribute queries. All services publish descriptions of themselves to SDS instances running in their local base. These instances are organized in a hierarchical structure, matching the administrative structure of the network. Summary information about known services is exchanged through this hierarchy; searches similarly propagate through the hierarchy until matching information is found.

3 Bases: Scalable Platforms for Internet Services

We have constructed a software platform that runs on cluster-of-workstation bases to help alleviate the challenges of scalable, high performance service construction. The platform consists of a programming model and I/O substrate geared towards obtaining high concurrency, and a cluster-based execution environment (**vSpace**) that provides facilities for service component replication, load-balancing, and fault tolerance. In addition, we provide services with a cluster-based, scalable storage platform (distributed data structures, or **DDSs**) that exposes a coherent image of persistent data across the physical nodes of a cluster. We describe each of these three elements in turn.

3.1 A Programming Model and I/O Substrate for High Concurrency Services

Popular Internet services must be able to handle a very high throughput, perhaps even reaching tens of thousands of requests per second in the extreme case. A service must remain robust under this extreme load, and it must also gracefully handle temporary bursts during which the offered load exceeds the capacity of the service. We call the process of achieving this robustness **conditioning the service**. A necessary (but not sufficient) step in conditioning is selecting an appropriate programming model and concurrency strategy that allows the service author and the service's execution environment to observe and manage constrained resources such as threads and client tasks.

Our programming model imposes a particular abstraction on services, illustrated in Figure 2. Given a request from a wide-area client, the service processes that request through a sequence of logically distinct stages, each of which is separated by a high- or variable-latency operation. For example, a web server might have three stages: reading and parsing an HTTP request from a browser, retrieving the requested file from the file system, and returning a formatted response to the browser. We impose the constraint that

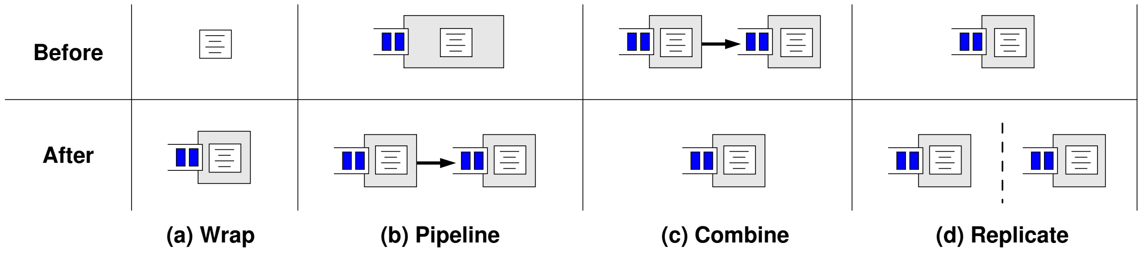

Figure 3: **The Four Design Patterns:** The four design patterns, **wrap**, **pipeline**, **combine**, and **replicate**, can be applied to stages of a service to condition it against load, failures, and limited or bottleneck resources.

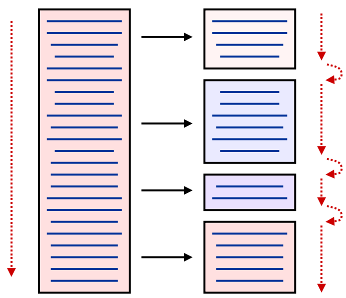

Figure 2: **Splitting a Service into Stages:** Our programming model views a service as a sequence of stages, separated by high- or variable-latency operations. Stages only share data using pass-by-value semantics, for example, by exchanging messages.

all data sharing between these stages is done using pass-by-value semantics, for example through the exchange of messages containing the data to be shared. This constraint acts to decouple the stages, allowing them to be isolated from each other, and perhaps be physically separated across address spaces or physical machine boundaries.

Given these separated stages, our programming model offers four **design patterns** that authors and the service infrastructure can apply to compose and condition these stages (Figure 3):

Wrap: The wrap pattern places a queue in front of a stage, and assigns some number of threads to the stage in order to process tasks that arrive on the queue. The queue conditions the stage to load; excess work that cannot be absorbed by the stage's threads is buffered in the queue. This queue also serves to expose scheduling and admission control mechanisms to the stage: because the queue is apparent, the code in the stage can decide the order in which to process tasks, and it can also choose to drop tasks in the case of excessive or long-lasting overload. Because threads are dedicated to the stage, applying the wrap pattern allows the stage to execute independently of other stages.

Pipeline: The pipeline pattern takes a wrapped stage, and splits it into two pipelined, wrapped stages. Pipelining further decouples a stage, and allows for functional parallelism across processors or cluster nodes. Pipelining permits optimizations such as having a single thread repeatedly execute the same code while processing many tasks from a queue, thereby increasing instruction locality.

Combine: The combine pattern is the logical inverse of the pipeline pattern. Given two previously independent, wrapped stages, the combine operator fuses the code of the two stages into a single, wrapped stage. Combine permits resource sharing and fate sharing between these previously independent stages.

Replicate: Given a wrapped stage, the replicate pattern duplicates that stage on a number of independent processors or cluster nodes. Replication is used to eliminate bottlenecks; by replicating a stage, the resources that can be applied to its bottleneck are augmented, hopefully increasing the throughput of the stage. Replication also duplicates the stage's functionality across multiple failure boundaries, introducing the potential for fault tolerance.

We have implemented a programming library that makes it simple for both service authors and the service's execution environment to apply these patterns to pieces of code. All network communication and disk I/O provided by this library is built using a non-blocking, asynchronous event-driven style of programming. This event-driven style nicely matches the task-driven composition of stages, and also permits each note to scale to the point where it can handle many thousands of concurrent tasks, network connections, and disk interactions.

3.1.1 Java-Based I/O Substrate Implementation

The Ninja base architecture makes extensive use of the Java [19] programming language, which provides strong typing, platform independence, code mobility, and automatic memory management. These language properties are greatly beneficial for engineering robust Internet services, eliminating many common sources of bugs. Java also provides flexibility in terms of service deployment across multiple architectures. We make use of optimizing Java compilers including OpenJIT [30] and the IBM JIT compiler [27].

Implementing the base platform in Java presented two important challenges. The first was the lack of nonblocking I/O mechanisms in the Java core libraries. We overcame this by implementing our own nonblocking I/O library using native code wrappers to existing system calls, for example nonblocking sockets and *select*. The second was providing efficient access to specialized interfaces, such

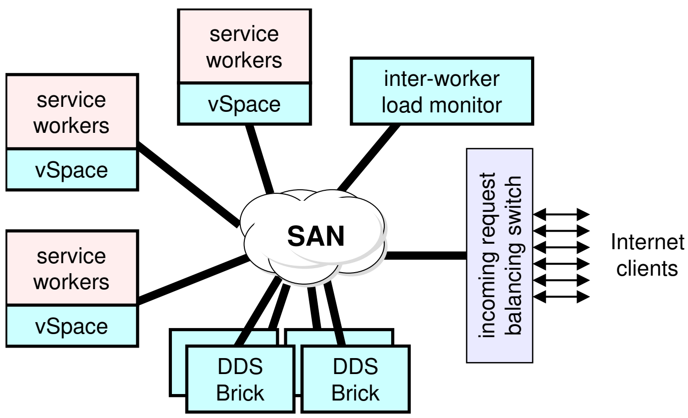

Figure 4: **Software Architecture of a Base:** A base consists of a cluster, the nodes of which run the vSpace execution environment. Services are implemented as a graph of workers which communicate through a typed task dispatch mechanism. vSpace load balances tasks across workers based on information from a cluster load monitor. Workers are replicated across nodes for scalability and availability, and share global state through a consistent, scalable storage platform (distributed data structures, or DDS).

as user-level network interfaces to the cluster's system area network. The native code interface provided by Java is ill-suited for these interfaces, as they require fast access to hardware resources and pinned I/O buffers outside of the Java heap. We have developed an extension to the Java environment, **Jaguar** [46], which performs a compile-time specialization of Java bytecode to perform low-level operations directly, while maintaining type safety and portability. We have used Jaguar to implement a Java interface to the VIA [7, 41] cluster network interface, which obtains 80 μs round-trip latency and over 488 Mbit/s bandwidth over the Myrinet [32] system area network. This is equivalent to the performance of VIA as accessed from C and is more than an order of magnitude greater than that possible when using Java's native code interface. Jaguar has also been used to implement fast object serialization and memory-mapped file access.

3.2 The vSpace Execution Platform

vSpace is an execution environment for scalable Internet services which operates on a cluster of workstations. vSpace services are constructed using the programming model described in the previous section; services are constructed as a graph of **workers**, each of which consists of a fixed-size thread pool, an incoming event queue, and a set of methods that implement the worker's logic. A vSpace service is described by a formal **service definition**, which precisely specifies the set of workers in the service, their code, and resource requirements. The act of **service publication** resolves intra-service dependencies and effectively "freezes" the code used by this particular version of the service. This allows the entire service to be treated as a versioned, immutable entity which is ready for deployment and composition with other services. Further modifications to the service code result in a new version of the service, and do not affect previously published versions.

Workers correspond directly to stages, as described in the previous section. Workers communicate by asynchronously pushing typed messages onto other workers' queues. Worker instances and workers of different types are pipelined, executing in parallel on the multiple CPUs and physical nodes in the cluster. vSpace uses the replicate design pattern to instantiate copies of workers across multiple cluster nodes; each worker instance, called a **clone**, uses the same code base and shares global persistent state through a distributed data structure (described below). A set of worker clones of the same type are called a **clone group**. vSpace automatically spawns and destroys clones in response to observed system load; workers' queue lengths and worker execution times are both used determine the current load. Scalability and fault-tolerance are obtained by replicating clones across multiple physical resources (such as the nodes of the cluster), and by providing a mechanism for failure detection and clone restart.

A worker may send one or more outgoing tasks to a named clone group, in which case the outgoing tasks are load-balanced across the clones in that group. Optionally, the sender may specify a particular clone as the destination for a task. This is used as a locality optimization, to allow the result of a previous task dispatch to return to the original sender.

3.3 Distributed Data Structures

A distributed data structure (DDS) [20] is a self-managing storage layer designed to run on a cluster of workstations at the scale required by Internet service workloads. A DDS has all of the previously mentioned service properties: high throughput, high concurrency, availability, incrementally scalability, and strict consistency of its data, but provides a narrow data structure interface. Service authors see the DDS as a conventional data structure, such as a hash table, a tree, or a log. Behind this interface, the DDS platform hides all of the mechanisms used to access, partition, replicate, scale, and recover the data in the DDS (illustrated in Figure 5). The DDS greatly simplifies service construction by hiding the complexity of robustly managing scalable persistent state that is partitioned and replicated across the cluster.

We have implemented a distributed hash table as an example DDS. All operations on elements inside this distributed hash table are atomic, in that any operation completes entirely, or not at all. The hash table has one-copy equivalence, so although data elements in the hash table are replicated across multiple hash table nodes (or **bricks**), workers that use the hash table see a single, logical data item. Two-phase commit is used to keep all replicas coherent. We have not yet implemented transactions across multiple elements or operations, but we believe that the atomic consistency provided by our current distributed hash table

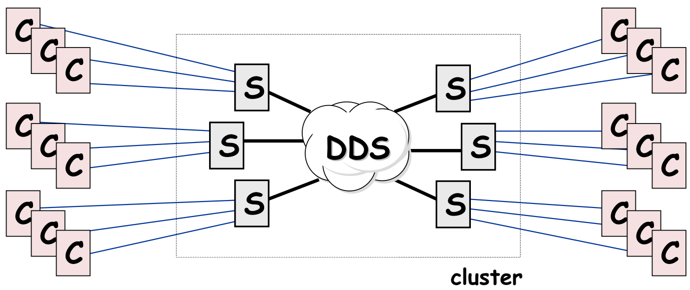

Figure 5: **High-level view of a DDS:** A DDS is a self-managing data repository running on a cluster of workstations. All service instances (S) in the cluster see the same consistent image of the DDS; as a result, any WAN client (C) can communicate with any service instance.

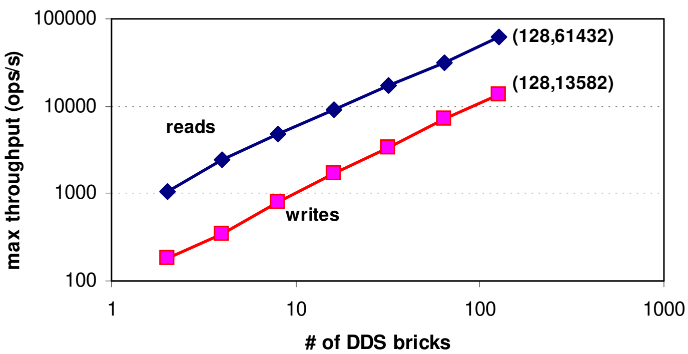

Figure 6: **Throughput scalability:** This benchmark shows the linear scaling of throughput as a function of the number of bricks serving in a distributed hash table; note that both axis have logarithmic scales. As we added more bricks to the DDS, we increased the number of workers using the DDS until throughput saturated.

is already strong enough to support a large class of interesting services.

To demonstrate the scalability and fault-tolerance of the distributed hash table, we have run a number of performance analyses on a large cluster of workstations (the UC Berkeley Millennium cluster [13], consisting of two hundred and twelve 500MHz Pentium CPUs across 67 SMPs, each with either 500MB or 1GB of physical memory, two 15 GB hard drives, and all connected by a Gigabit switched Ethernet). Figure 6 demonstrates linear scaling in throughput of the distributed hash table as the number of brick nodes serving data is increased; note that for this experiment, most data was resident in a physical memory cache on brick nodes, rather than forcing a read from disk per request.

In Figure 7, we show the read throughput of a 3-node distributed hash table as a fault is deliberately induced in one node, and as that failed node undergoes recovery. This figure shows that the read throughput of this hash table degrades to 2/3rds of its initial throughput as one of the three

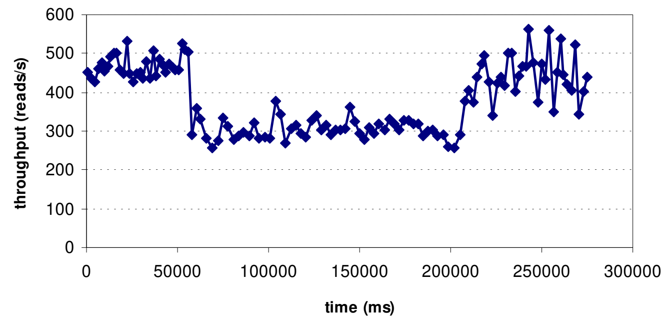

Figure 7: **Availability and Recovery:** This benchmark shows the read throughput of a 3 node hash table as a deliberate single-node fault is induced, and afterwards as recovery is performed.

nodes crashes, but quickly resumes to its full throughput as the crashed node completes its recovery.

We have also experimented with scaling the capacity of a distributed hash table by creating and populating a single hash table with over 1 terabyte of data spread over 128 CPUs and 128 disks. This 1 TB hash table took 1.5 hours to populate, achieving a write throughput of 256 MB/s (2 MB/s per disk). The disk write performance was seek limited, as random keys were inserted into the hash table for this experiment.

Our DDS implementation makes use of the exposed queues and events (as described in Section 3.1) to implement efficient internal task scheduling. Exposing the queues to the DDS code makes it is possible for each DDS brick to "peek" into its queue of incoming requests and schedule them based on resource availability. For example, incoming read requests for which data is present in the buffer cache can be scheduled before those requiring disk access. This technique leads to higher throughput, as head-of-line blocking is reduced. The use of event queues also makes it possible to reorder disk accesses for greater locality and to perform prefetching, similar to optimizations used in filesystems and database storage managers.

4 Units

The space of units is extremely diverse with large variations in CPU, memory, and storage capabilities, communication bandwidths and latencies, and user interfaces. In this section, we briefly circumscribe this space by describing a representative set of units. We then focus on a particularly interesting new class of units, networked sensors, that are the most constrained in terms of capabilities and resources.

PCs and laptops are examples of extremely capable units, in that they have liberal amounts of CPU and memory resources, persistent storage, and sophisticated display capabilities. However, laptops still must be capable of dealing with mobility, disconnected operation, and low bandwidth or unreliable communication over wireless networks.

Figure 8: **A TinyOS-based Mote:** This "mote" includes a 4MHz microcontroller, a software-driven radio, and an application that coordinates with neighboring motes to discover an ad-hoc sensor network routing topology.

PDAs represent a class of device with limited computation, displays, user interfaces, and persistent storage. Cell phones are currently distinct from PDAs in that they have much more limited computational abilities and they are essentially continuously connected to the network. There is, however, a strong trend towards the convergence of PDA and cell phone capabilities, yielding a class of units that has the minimal graphical user interface, storage, and programmability of a PDA, but with the continuous connectivity of a cell phone.

The most limited form of units that we consider are networked sensors and actuators. These devices have extremely limited computational resources, almost no storage capabilities and no human interface. In addition to limited communication bandwidth, communication is extremely expensive for these devices, since power is their most critical resources and communication consumes significant power. As an example of networked sensors, we have developed a device, called a **mote**, that contains a microcontroller, a radio, and photo and light sensors. This device (which is slightly larger than a quarter) can be placed in the environment or carried by an individual, and reports information collected from its sensors to services for analysis. We report further on our experiences with this device below.

4.1 Characteristics of Networked Sensors

The characteristics of networked sensors require a design methodology focused on extreme efficiency, both in terms of computation and power. As an example networked sensor, we have assembled a "mote" that includes an ATMEL 8535 4MHz microcontroller with 512B of SRAM and 8KB of flash memory, an RF radio with 10kbps throughput, a light and temperature sensor, and three LEDs for visual feedback of information (Figure 8).

Somewhat surprisingly, the programming model that we have designed for these tiny devices is very similar to that of high throughput services in vSpace, although we use this model for the sake of power and computational efficiency, rather than throughput and load conditioning. Power is the most precious resource on these devices, and communication is the most expensive operation in terms of power consumption. Given this, the ability to put hardware components into a standby state can save significant amounts of power. To maximize the opportunity for putting the device's CPU into standby, we have developed an event-driven software architecture for our operating systems and applications. The presence of blocking operations could limit the system's ability to switch into a low power mode, especially if hardware polling is used to complete a blocking operation. In contrast, with an event driven system, all processing occurs in response to hardware events. This allows the processor to enter standby mode between events, as there no computation needs to be done until the next hardware event occurs.

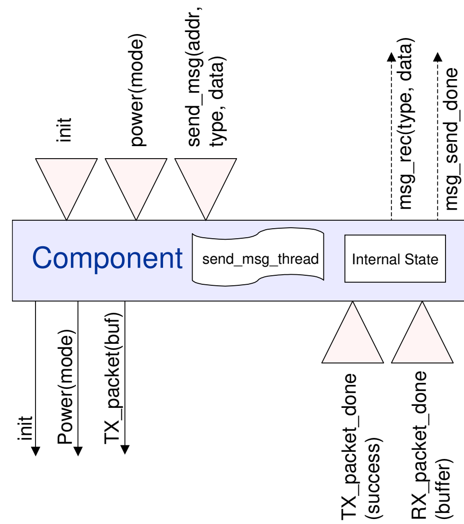

Figure 9: **A TinyOS Software Component:** Each TinyOS software component accepts and emits commands and events. Commands flow from higher level layers to lower levels, and events flow from lower level layers to higher levels.

We have also observed that our network sensors must be able to handle significant amounts of concurrency. Sensors are typically I/O centric, and must be capable of supporting multiple, simultaneous flows of information. Flows can be local to a sensor (e.g., the interaction between a CPU and the physical sensor devices or the radio used for communication), or they may span across multiple sensors in a sensor network. For example, networked sensors may cooperate to propagate each other's data towards a central collection point. In this case, the microcontroller's interaction with its sensors must be overlapped with its operation of the radio and execution of networking protocols. To exacerbate the situation, on many microcontrollers the CPU must directly interact with the radio (compared with PCs which typically have dedicated NICs to service the communications device), introducing real-time constraints.

To address these challenges, we have developed the TinyOS operating environment for networked sensors. TinyOS has a component-based architecture in which each hardware and software component exports an interface that contains the set of commands that it accepts as well the set of events that it fires (Figure 9). Internally, a software component is given a statically allocated storage frame. While handling a command, a component can emit tasks that TinyOS's scheduler must execute. Tasks are similar to vSpace workers, but they share the state of the component that created them rather than sharing state through the passing of typed Java objects. Unlike general purpose threads, TinyOS tasks execute to completion and are atomic with respect to each other. TinyOS includes a two-level scheduling mechanism that allows high priority events to preempt low priority tasks. Real-time constraints (e.g., servicing the radio) are met by using high priority events, while less critical operations (such as gathering data from a temperature sensor) are serviced with the remaining low priority CPU time.

The use of the TinyOS component model and scheduler greatly simplifies the composition of multiple components on a sensor. To demonstrate this, we have build an application in which our motes self-assemble in an ad-hoc network, and communicate their routing information to a statically configured active proxy node. This routing discovery application, as well as the particular operating system tuned for this hardware, are composed of several TinyOS components. The components are composed together using a CAD tool (which represents commands and events with CAD symbols), and structural VHDL is exported by the CAD tool. This VHDL is used at compile time to assemble the system image that is downloaded into the device's flash memory. A particularly interesting feature of these devices is the ability to wirelessly reprogram them by sending system images over the sensor network.

5 Active Proxies

Active proxies in the Ninja architecture serve the purpose of performing "impedance matching" between client devices and services by adapting data and access protocols to the devices' and services' needs. Because active proxies can execute in an environment local to devices, active proxies can perform context-aware optimizations and transformations on behalf of devices. We believe that active proxies bring three essential properties to the Ninja architecture: dynamic service adaptation, secure access to information, and the fusion of multiple devices. We describe each of these in turn.

5.1 Dynamic Service Adaptation

In the Ninja architecture, active proxies assume the responsibility for mitigating the heterogeneity of units by translating both network protocols and data formats between clients and services. At the network protocol level, active proxies can communicate with clients through protocols specially designed for low computation, low power, or poorly connected devices. This is important since common service communication protocols such as Java RMI, Ninja RPC, and Jini assume that clients are well connected and computationally powerful. Similarly, active proxies can be used to help establish connections between clients and services by performing more complicated tasks associated with cryptographic handshakes [16].

Additionally, active proxies can distill service content into a format more suitable for small devices [17]. Content presentation can be tailored for small screen layouts, and image resolution and bit-depth reduced both for limited display and network capabilities of these devices [15, 17]. For example, a HTML representation of the content can be rendered as WML for a WAP [44] enable phone, a custom application format, or even as voice. Active proxies may perform this filtering at the application level (e.g., by selectively dropping MPEG frames in a video stream), or at the protocol level (e.g., by delaying or compressing data to increase actual or perceived throughput, based on knowledge of the network conditions that the device is currently experiencing).

5.2 Secure Service Access from Diverse Clients

Current security models of infrastructure services assume that both the user's access device and the software running on it can be trusted not to intercept or send private information elsewhere. Unfortunately, this is not the case for many access points, including public kiosks. A subverted kiosk is able to record all keystrokes (such as typed passwords), monitor network traffic to extract personal information such as account numbers or mailing addresses, or perform active attacks by hijacking connections, even if the network transmission is encrypted. To avoid such attacks, trusted active proxies can perform context-aware transformations on data before it arrives at a kiosk to reduce the content value. Proxies can also introduce alternative authentication mechanisms (such as one-time passwords) so that users will not need to divulge passwords or other personal information to untrusted infrastructure. In Section 8.2, we describe in detail an example framework that has this functionality.

PDAs are problematic because they are generally power-constrained, computationally-limited devices with little memory and poor networking capabilities. To perform the industry-standard SSL handshake phase on one such device (a Palm Pilot) requires 5-10 seconds. This latency imposes an intolerable delay for connection setup, which is particularly undesirable if network connectivity is intermittent. An SSL implementation that uses elliptic curve cryptography [8] is feasible on a Palm Pilot V, but few Internet services support that option. Active proxies can be used to adapt the security requirements of services to the capabilities of the device. Trusted active proxies can present units with power and computation efficient security protocols, while communicating with end services through standard protocols.

5.3 Multiple Device Fusion

In addition to enabling basic access, active proxies can be used to combine the capabilities of several devices. This is useful for both content and security adaptation. For example, the limited GUI of a PDA can be supplemented by the richer, larger display of a public kiosk, by placing most of the application on the kiosk while displaying and entering sensitive personal information on the PDA. Entering form data using a pen-based interface is tedious at best and even more cumbersome using number pads on devices such as cellular telephones. Active proxies can split the trust between the PDA and the public terminal by fusing the devices together to provide one logical channel with secure access to the end service. This device fusion can only be done because the active proxy is aware of the context in which the devices are being used, and thus serves as another example of context-aware adaptation.

6 Service Location across the Wide Area

The service discovery service (SDS) [10] serves two important, and complementary roles: it provides a mechanism by which services can announce their presence to the infrastructure, and it provides a mechanism by which both human users and programs can locate these announced services across the wide area. While designing the SDS, we focused on providing a fully secure, semantically rich service location system that would successfully scale to the wide area. The SDS is a scalable, fault-tolerant, and secure information repository, providing clients with directory-style access to all available services. Services *describe* themselves to local SDS instances; these descriptions are *published* and aggregated across a wide-area hierarchy, and clients can *query* this hierarchy of SDS instances in order to locate services.

In addition to serving as a location mechanism, the SDS also plays an important role in helping clients determine the trustworthiness of services, and vice versa. This role is critical in an open environment, where there are many opportunities for misuse, both from fraudulent services and misbehaving clients. To address security concerns, the SDS controls the set of agents that have the ability to discover services, allowing capability-based access control, i.e., to hide the existence of services rather than (or in addition to) disallowing access to a located service.

As a globally distributed, wide-area service, the SDS must surmount challenges that are not faced by services that operate solely inside a base. The global SDS service must be robust against network partitions and component failures, it must address the potential bandwidth limitations between remote SDS entities, and it must arrange its individual SDS instances components into a hierarchy to distribute the query workload (implying queries must be routed across this hierarchy).

6.1 Design

The SDS system is composed of three main components: clients, services, and SDS servers. Clients want to discover the services that are running in the network. SDS servers solicit information from the services and then uses it to fulfill client queries. To provide scalability in both number of services and volume of client requests, SDS servers are organized into a hierarchical structure. Services and requests are associated with SDS servers according to each server's **domain**, the network extent that it covers.

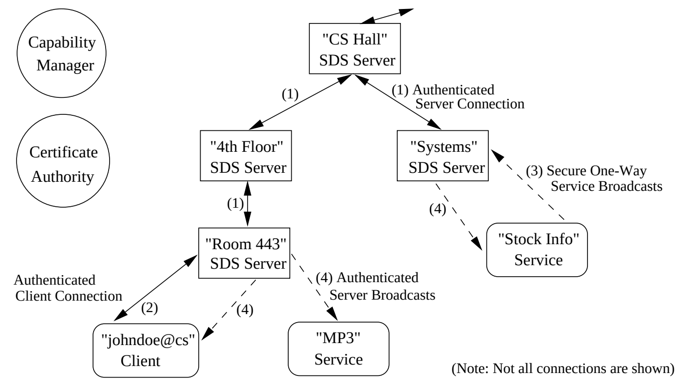

Figure 10: **Components of the Berkeley SDS:** Dashed lines correspond to periodic multicast communication between components, while solid lines correspond to one-time transport connections.

To propagate information across potentially heterogeneous service architectures, we use an "announce/listen" model of data propagation. Servers cache data contained in periodic multicast messages. Component failures are tolerated through the course of normal operation, removing the need for a separate recovery procedure: recovery is enabled simply by listening to channel announcements [2].

To provide both flexibility and simplicity in the service query mechanism, the SDS uses XML to encode both service descriptions and queries. XML allows the encoding of arbitrary structures of hierarchical named values, and supports validating service descriptions against well-defined schemas (Document Type Definitions). This mechanism gives SDS servers functionality to validate select service descriptions while allowing evolution of existing service description schemas.

SDS servers are responsible for sending authenticated messages to the well-known global SDS multicast channel, including announcing multicast addresses to be used for service announcements, the rate at which announcements should be repeated, and contact information for the Certificate Authority and the Capability Manager. On each channel, a measurement-based periodicity estimation algorithm determines the optimal send rate for messages that produces the desired tradeoff between update latency and bandwidth utilization. Each server can spawn a child server in order to hand off a portion of its load. Parents monitor child servers through heartbeat packets, and will restart a crashed child server. Because servers keep a cache of announcement service descriptions, a restarted server restores its data by listening to the channel.

To provide authentication, privacy, and access control, SDS servers work with a Certificate Authority (CA) and a Capability Manager (CM) using secure communication protocols. The CA is a trusted source which provides proof of the binding between a principal and its public and encryption keys, in the form of a certificate. The CM manages individual access control lists (ACLs) on behalf of each authenticated service. Communication between SDS components utilize appropriate security measures while minimizing the performance penalty. SDS server announcements need authentication, and are therefore signed, including an embedded timestamp. Service providers encrypt their description broadcasts with a symmetric key, which accompanies the message as a data block encrypted by the server's public key. This allows caching of the symmetric key in the common case, and simple recovery of the symmetric key during failure recovery. Communication among servers and clients relies on a separate authenticated transport channel.

A client queries its SDS server over an authenticated channel to pass in a XML service template and its access rights in the form of capabilities. The server uses an internal XML database (called XSet [51]), to find services accessible to the client which satisfy the query, and returns them to the client.

In order to make their services known, service providers listen on the global multicast channel to routinely determine their current responsible SDS server. Providers periodically broadcast service descriptions to a multicast channel, using the address and broadcast rate defined by the server announcement messages. Providers are also responsible for contacting a capability manager and defining access control information for its services.

6.2 Wide-area Operation

The SDS wide-area hierarchy is designed to scale up in both query volume and number of available services, while adapting to changes in the underlying entities. The primary goal is to allow queries from all clients to reach services on all SDS servers. In our approach, servers dynamically arrange themselves into a multi-level hierarchy, summary information is propagated up to parent servers, and queries are partitioned among and forwarded to the relevant servers.

The actual organization of the hierarchy can be dependent on many criteria, such as administrative domains or network topology. We believe that the mechanism should support the existence of multiple hierarchies, and actual usage should be based on policy. Individual servers can choose to participate in more than one hierarchy by keeping multiple routing tables, one for each hierarchy.

To prevent upper-level servers in the hierarchy from being overwhelmed by update or query traffic, the SDS architecture filters information while it propagates upward. In particular, the information is summarized in a way that allows queries to determine which, if any, branches contain potential matches.

To accomplish this lossy aggregation, we use hash summarization, where information is summarized using a unique N-to-M mapping of data values. Complicating this procedure is the SDS' use of the subset query model, where matching documents can be identified by a partial list of service characteristics. Our solution is to hash a limited number of tag subsets, each subset containing a single tag or a cross-product of two tags. This limits computation required for summarization. To address the issue of storage space for summarizations, we use Bloom filters [5]. Bloom filters collapse hashed summarizations into a fixed-size table, accepting greater possibility of false positives in return for less storage requirements.

Description	*Latency*
Query Encryption *(client-side)*	5.3 ms
Query Decryption *(server-side)*	5.2 ms
Authenticated Transport Overhead	18.3 ms
Query XML Processing	9.8 ms
Capability Checking	18.0 ms
Query Result Encryption *(server-side)*	5.6 ms
Query Result Decryption *(client-side)*	5.4 ms
Query Unaccounted Overhead	14.4 ms
Total (Secure XML Query)	82.0 ms

Table 1: Secure Query Latency Breakdown

In summary, SDS servers dynamically organize themselves into potentially multiple hierarchies for data partitioning and query routing. Each servers uses multiple hash functions on various subsets of tags in service announcements, and uses the results to set bits in a bit vector. Servers which are internal nodes in the hierarchy combine bit vectors from itself and its children servers, and associates the result with this branch at its parent node. After receiving a query, each server checks its own bit vector for a match, and failing that checks its children vectors to determine which branch to forward the query to. A server resolves a query against the vector by multiply hashing it and checking if all the matching bits are set in the bit-vector. A missing bit guarantees a true miss, while a match could signal either a false positive or a true hit.

The distribution of data across the wide area exposes a trade-off between consistency and performance. Strict consistency is difficult to achieve in the face of frequent updates, given the wide area's constraints on network bandwidth, transmission latency, and the greater possibility of network partitions. Therefore, the SDS system provides loose consistency guarantees about service location information across the wide area.

6.3 Performance

We have measured the performance of a single SDS server on an Intel Pentium II 350Mhz with 128MB RAM, running on Linux 2.0.36 using the Blackdown JDK 1.1.7 and the TYA JIT compiler. The results are presented in Table 1. This table shows that the primary sources of latency are the authenticated transport connections and capability checking using the Cryptix Java security library. We expect both of these components will decrease significantly as a result of ongoing research. Furthermore, the XML query processing is shown to scale logarithmically with the size of the data set [51]. Finally, using these performance numbers, we estimate that a single SDS server (using off-the-shelf components) can handle a user community of about five hundred clients sending queries at a rate of of one query per minute per client.

7 Paths: Composition of Services Across the Wide-Area

The primary goal of paths is to facilitate the composition of services. To be most useful, the infrastructure should attempt to automate as many parts of the path creation process as possible. In our design, an automatic path creation (APC) facility automates the task of finding paths between system components, creating the network connections between components, fine tuning the performance of the data flow, and handling error conditions. Whenever possible, the APC facility protects users from the failure of individual path components or communication links. The ideal situation would be to provide the illusion that the user is accessing a single robust service providing the composed functionality. Because the APC facility handles large numbers of concurrent users, we designed its path construction algorithms to scale well as the number of components increases, even though the number of possible paths may grow exponentially as components are added.

A path is comprised of a sequence of **operators** that perform computations on data and **connectors** that provide protocol translations between operators. A connector is a channel through which operators can pass application data units (ADUs). The connector hides potential differences in network protocols from the operators, and allows them to communicate as long as the output data type of the downstream operator matches the input data type of the upstream operator. Each connector is characterized by a specific transport protocol.

Operators perform computation on data flowing along the path. Operators are strongly-typed: they have a clear definition of the input they accept and the outputs they produce. Operators have various attributes such as supported communication protocols, computational requirements, or required external data (e.g. a remote database). In addition, operators have associated cost metrics, which describe the run-time performance of the operator and are used for optimization during path creation. The type and attributes for each operator are combined to form an XML description of the operator. These descriptions are used to determine which combinations of operators could make a valid path.

The Ninja architecture provides two classes of operators: long-lived and dynamically created. Long-lived operators are standard Ninja services, and hence support both the data persistence and fault-tolerance properties previously discussed for services. These are registered with and located through the service discovery service (SDS). Dynamically created operators are light-weight, short-lived transformation elements created by the APC facility as required. These operators, which run in active proxies, have only soft-state and hence can be simply restarted if the active proxy fails.

While the reliability of both long-lived and dynamic operators helps to guarantee that a path can be reconstructed when a failure occurs, this does not safeguard against the loss of data that was already in the path when the failure occurred. Hence, applications that use paths must provide their own mechanisms for guaranteed or in-order data delivery if this is required.

7.1 An Example of a Path

As a motivating example, consider a map service that provides driving directions in response to a user-specified address. This example illustrates the composition of two operators with a service, and shows how active proxies that are selected by the path creation process are used to perform protocol and data format translations between clients and services. To allow access to the overall audio driving direction service from a cell phone, the APC facility might create a path as follows:

1. The user initiates a call from a cellular phone. The user speaks the address to which she wishes to get driving directions. An RTP-based audio connector is used to send this audio to the first operator in the path.

2. A speech-to-text operator, running in an active proxy, is used to convert the spoken audio into structured text using a grammar specifically chosen for this context (address input). The structured text emitted from this operator is passed along a TCP-based reliable bytestream connector to a map service.

3. A map service, running in a base, receives the address, and returns structured text representing driving directions to the specified address. These directions are passed along a TCP-based reliable bytestream connector to the next operator in the path.

4. A text-to-speech translator, running in the same active proxy as the speech-to-text operator, transforms the textual driving directions into audio. An RTP-based audio connector is used to send this audio to the user's cell phone.

5. The user hears the driving directions being spoken to her over her phone.

7.2 Path Construction

To create a Ninja path, a user provides the APC facility a specification of the endpoints of the required path, a partially ordered list of operators that must be included in the path, and an acceptable range of costs for the path in terms of latency, computation or memory requirements. This information is used to construct an optimal path for the user's specific requirements. The path construction process consists of four steps. As shown in Figure 11, path construction is a process of continuous feedback and optimization. The details of each step are described below.

Step 1: Logical Path Creation: A logical path consists of an ordered sequence of operators that are joined with

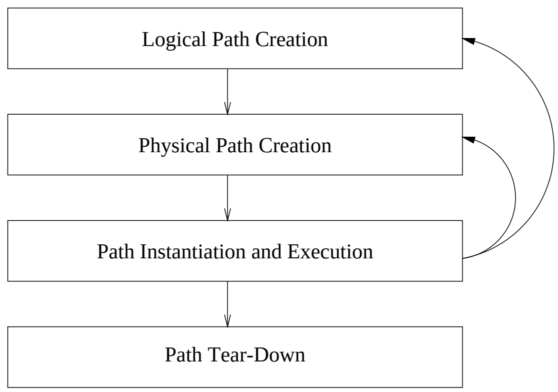

Figure 11: **Path construction process:** Path execution is an iterative process of optimization. The Ninja APC facility guarantees the availability and fault-tolerance of a constructed path by rebuilding its physical or logical path.

connectors. During logical path creation, the APC facility searches through the XML descriptions of the operators, to find valid sequences that could perform the computation requested by the user. The result of logical path creation is a list of possible operator sequences. Note that since some operators may be commutative (image format transcoders, for example), the space of all possible logical paths is large. Hence, only a small number of logical paths are considered initially.

Step 2: Physical Path Creation: A physical path is a mapping of a particular logical path onto physical nodes which execute the operators. Nodes for long-lived operators are chosen from the known services that provide the desired functionality, as located using the SDS. Nodes for short-lived operators are chosen according to the computational capabilities of the node, and the cost of using that node in the path. The APC facility constructs a physical path from a logical path by finding the lowest cost nodes that meet the user's requirements.

Step 3: Path Instantiation, and Execution: Once the nodes of the path have been selected, the APC facility starts any required dynamic operators, and sets up appropriate connectors between the various operators. Once all nodes in the path are set up, data flow is started. In addition, a control channel (used for reporting of error conditions and performance information) is established between the operator nodes and the APC facility. During the lifetime of the path, the APC facility actively monitors the operator nodes to make sure that they are functional. Operator nodes report problems to the APC facility about their neighboring nodes in the path, so that the path is repaired when necessary. The APC facility monitors the performance of the path, and re-routes the data flow if new conditions make the original path suboptimal. The control path is used for exception handling, controlling parameters of path components, monitoring and analyzing path performance; thus, it needs to be independent of data paths and be highly robust.

Step 4: Path Tear-Down: When a path is no longer needed, the user informs the APC facility that it should be removed. The APC facility then stops the data flow, removes connectors, and shuts down any dynamic operators. As a performance optimization, the APC facility may cache commonly used logical and physical paths for re-use at a later time.

7.3 APC Implementation and Evaluation

We have developed an initial prototype of the APC facility that supports both long-lived and dynamic operators. In addition, we have a special class of dynamic operators that can be used to wrap existing services. This allows the APC system to make use of older services that can't communicate directly with our connectors.

Each operator has a reference to an output and input connector that speaks a specific transport protocol. All connectors implement a common Java interface. To interact with previous and subsequent operators in the operator chain, each operator invokes read and write methods of this interface to receive its input data and send its output data. TCP, UDP, and RTP connectors are supported in the current prototype.

Our current implementation encompasses the full range of path creation described previously. Logical paths are created by searching the XML descriptions of the available operators to find the smallest number of operators that can perform the desired data flow. A physical path is then selected by placing operators on the least loaded nodes of the network.

Machine failures are automatically detected by the APC service, and running operators are restarted on other nodes. Fault detection is achieved by either time-out of a heartbeat beacon or by catching an I/O exception when reading or writing data from or to the failed machine. Our prototype does not presently exploit the possibilities for performance tuning through dynamic reconstruction of paths.

8 Example Services

Having completed the description of the Ninja architecture, in this section of the paper we describe a number of interesting applications that we have built on top of it. These applications demonstrate the capability of the Ninja architecture to facilitate the simple construction of robust, scalable services that are accessible by a diversity of devices. This illustrates the opportunity that our architecture provides for the widespread innovation of both services and devices.

8.1 The Ninja Jukebox

The Ninja Jukebox [18] was an early application built using our architecture, and it demonstrates some of our platform's key features. The Jukebox allows a community of users to build a distributed repository of digital music, and provides a collaborative filtering mechanism based on users' music preferences. Cluster nodes are harnessed to rip MP3 files from their local CD-ROM drives, and to act as servers for streaming MP3 to clients. One node acts as the music directory, and maintains a soft-state index of the songs published by each cluster node; the Jukebox client application contacts the directory to obtain a list of songs, and streams MP3 directly from the appropriate node using HTTP.

The Ninja Jukebox is based on MultiSpace [21], an early design prototype of the base service platform. MultiSpace nodes, each running a JVM, communicate through the use of NinjaRMI, an extensible variant of Java Remote Method Invocation [38]. Each component in the Jukebox application exports a NinjaRMI interface which is invoked either internally to the cluster or externally by the Jukebox client application (which also makes use of NinjaRMI). NinjaRMI provides support for strong authentication and encryption, which is used to control access to the Jukebox service. Each song in the Jukebox can have an associated ACL authorizing a particular set of users to listen to it.

Constructing this application as a set of strongly-typed, distributed components greatly simplified service construction and facilitated evolution, as new components could be added to the service as needed. An example of service evolution was our addition of the Jukebox Query Engine [45], which allows users to search for music in the Jukebox based on musical similarity between songs. The user provides a query song and a set of parameters to use for the search, as well as the number of results to return; the Query Engine returns the songs in the Jukebox which sound the most similar to the query song. The search is based on a k-nearest-neighbor search in a multidimensional space of features previously extracted from each song. The Query Engine runs on the same MultiSpace platform as the Jukebox itself, and its user interface is integrated into the Jukebox client.

8.2 An Active Proxy Framework for Accessing Services through Untrusted Devices

A more general service that we have implemented is an active proxy framework that provides secure multi-modal access to Internet services from units [23]. Consider the case of users accessing their stock trading accounts from public access terminals. Instead of relying on the terminal to protect their secure information, the users can direct private or sensitive information such as portfolio values or account numbers to their personal PDA, while using the rich GUI capabilities of the public terminal to initiate requests and display generic stock information (e.g., stock price fluctuations and historical graphs). Users initiate trading operations through the untrusted public terminals, but then confirm them using their trusted portable devices. Network connections to the users' PDAs are provided either by the environment, such as with kiosks with infrared network connections, or by the devices themselves , for example by directly initiating a connection from a wireless data enabled PDA.

The proxy is implemented as a collection of vSpace workers that abstract the functionality of security adaptation,

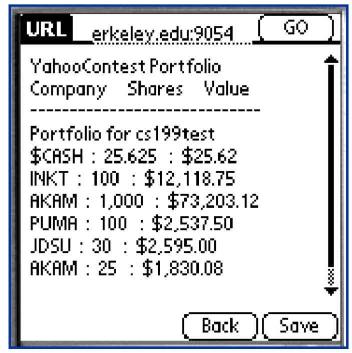

Figure 12: **WML for PDA:** WML holdings page customized for PDA display.

service adaptation, and device fusion. By combining generic content and security transformation functions with service-specific rules, the proxy architecture decouples device capabilities from service requirements and simplifies the addition of new devices and services. The service uses XML as a standard data representation; one vSpace worker transforms requests from the untrusted access device into an XML representation. Another worker provides access control filtering on these requests, possibly injecting secure information into the request: for example, this worker may convert a one-time password provided by the user through the untrusted terminal into a password that must be supplied to the service being accessed.

A third worker transforms the request from its generic XML representation into whatever protocol is necessary to access the service. For example, if the service is web-based, this worker will convert the XML into an HTML form to be submitted to the service's web server. This worker receives the content returned from the service, and transforms it back into XML. A fourth worker performs a sequence of filtering operations on the data in order to remove any sensitive information that should not be revealed to the untrusted device. A final worker is used to transform this filtered XML into whatever protocol and data format is needed to render the content on the untrusted terminal. This final XML transformation is driven using device-specific XSL style sheets.

The framework currently allows access to both the Datek Online [12] and YahooContest [48] stock trading services, and we are currently adding access to a HTML based mail service. Adding support for a new service merely requires authoring a script to convert the service's content into an XML representation. For example, the YahooContest service consists of half a dozen scripts each of which are approximately 250 lines or less. Rendering content for different client formats is requires authoring the appropriate XSL style sheets. Our example stock services consist of half a dozen style sheets for each device format, each ranging from 50–300 lines in length. Figure 12 shows the output of the YahooContest service rendered as WML for a WAP browser running on a trusted Palm Pilot. In this example,

Welcome, |||||||||

Portfolio for |||||||||

<table>
<tr><th>symbol</th><th colspan="2">last trade</th><th>change</th><th>volume</th><th>shares</th><th>value</th><th colspan="2">value change</th><th>paid</th><th colspan="2">gain</th></tr>
<tr><td>$$CASH</td><td></td><td></td><td></td><td></td><td>||||||</td><td>||||||</td><td>-</td><td>0.00%</td><td>|</td><td>-</td><td>-</td></tr>
<tr><td>INKT</td><td>4:00PM</td><td>119 3/16</td><td>-2</td><td>2,343,500</td><td>|||</td><td>||||||||||</td><td>-$200.00</td><td>-1.65%</td><td>||||||||</td><td>-$310.00</td><td>-2.54%</td></tr>
<tr><td>AKAM</td><td>4:01PM</td><td>70 1/2</td><td>-2 45/64</td><td>1,601,500</td><td>|||||</td><td>||||||||||</td><td>-$2,703.12</td><td>-3.69%</td><td>|||||</td><td>-$9,760.00</td><td>-12.16%</td></tr>
<tr><td>PUMA</td><td>3:59PM</td><td>25 7/16</td><td>+1/16</td><td>898,500</td><td>|||</td><td>|||||||||</td><td>$6.25</td><td>+0.25%</td><td>||||</td><td>$83.75</td><td>+3.40%</td></tr>
<tr><td>JDSU</td><td>4:00PM</td><td>86</td><td>-1/2</td><td>12,736,900</td><td>||</td><td>|||||||||</td><td>-$15.00</td><td>-0.58%</td><td>||||||</td><td>$53.75</td><td>+2.13%</td></tr>
<tr><td>AKAM</td><td>4:01PM</td><td>70 1/2</td><td>-2 45/64</td><td>1,601,500</td><td>||</td><td>|||||||||</td><td>-$67.58</td><td>-3.69%</td><td>||</td><td>-$247.50</td><td>-12.31%</td></tr>
<tr><td colspan="6">Totals(USD):</td><td>||||||||||</td><td>-$2,979.45</td><td>-3.23%</td><td>||</td><td>-$10,180.00</td><td>-10.23%</td></tr>
</table>

Figure 13: **Security Filtered HTML:** Stock holdings filtered for display on a public kiosk.

because the device is trusted, sensitive information such as the number of owned shares has not been removed by the proxy. In Figure 13, we show the output of the Datek trading service rendered as HTML on an untrusted web browsing kiosk; note in particular that sensitive information such as account numbers and the number of purchased shares have been removed.

8.3 NinjaMail

Electronic mail was one of the "killer apps" of the early Internet, and even today, the number of users with e-mail access is growing exponentially. At the same time, these users are expecting more complex functionality such as embedded multimedia and anytime/anywhere access. These two trends have implications on the requirements of modern e-mail servers. Hotmail alone has over 61 million active users [33], and if they offered just 50 MB worth of storage to each user, their servers would have to handle over 3 petabytes of data.

The goal of the NinjaMail [42] project is to build a scalable and feature-rich e-mail service on top of Ninja. NinjaMail was built to act as a general e-mail infrastructure which other applications and services could use to provide more specific functionality, as depicted in Figure 14. This loose coupling of the separate components allows for more flexibility and extensibility than traditional email servers.

At NinjaMail's core, the MailStore module handles storage operations such as saving and retrieving messages, pushing out notification of email events, updating message metadata, and performing simple per-user message metadata searches. A message's metadata represents its mutable attributes which are used to record its flags and current folder. Access modules support specific communication methods between users and NinjaMail, including an SMTP module for pushing messages into the MailStore and POP and HTML modules for user message access.

Each of the above modules is a separate worker running in the cluster, with scalability being achieved by running multiple clones of the worker. We found that decomposing the NinjaMail system into a set of workers to be a natural programming model and the typed task dispatching allowed

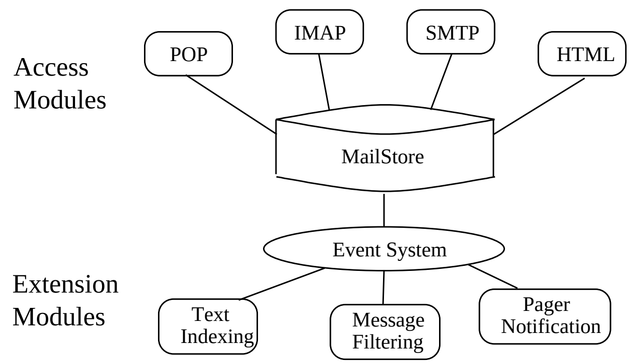

Figure 14: The NinjaMail architecture.

the components to be easily composed. We also created an event mechanism that allowed extension modules to register with the MailStore service to receive notifications when particular events occur such as e-mail receipt. This allowed for very diverse services to be built, such as an instant messaging notifier of new email.

8.4 Sanctio

Recently, there has been an ongoing controversy over access rights to proprietary instant messaging networks, such as AOL's AIM network [1]. Many companies have tried to compose their own services with these existing networks, however the owners of the proprietary networks have attempted to prevent such composition, as it diminishes their perceived market penetration.

We have built a service, called Sanctio, which is an instant messaging gateway that provides protocol translation between popular instant messaging protocols (such as Mirabilis' ICQ and AOL's AIM), conventional email, and voice messaging over cellular telephones. Sanctio obviates this controversy by bridging together these previously proprietary networks into an instant messaging internetwork. Sanctio runs on a vSpace base, and acts as a middleman between all of these messaging protocols, routing and trans-

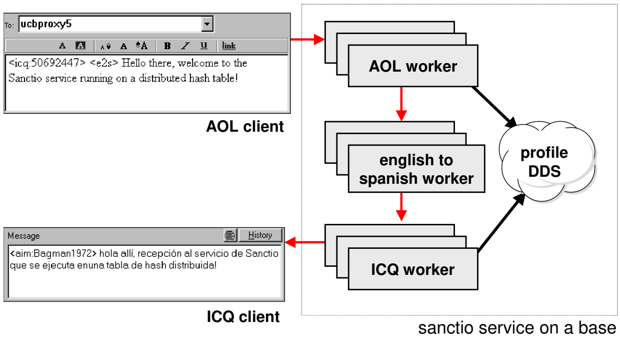

Figure 15: **Sanctio Messaging Proxy:** the Sanctio messaging proxy service is composed of language translation and instant message protocol translation workers in a base. Sanctio allows unmodified instant messaging clients that speak different protocols to communicate with each other; Sanctio can also perform natural language translation on the text of the messages.

lating messages between the networks (Figure 15). In addition to protocol translation, Sanctio also can transform the content of messages. We have built a "web scraper" that allows us to compose AltaVista's BabelFish natural language translation service with Sanctio, and thus the service can perform language translation (such as English to French) as well as protocol translation. A Spanish speaking ICQ user can send a message to an English speaking AIM user, with Sanctio providing both language and protocol translation.

Users can take advantage of unmodified commercial client application software in order to use Sanctio, or they can use software that we have constructed for mobile devices such as Palm Pilots. This software interacts with the Sanctio service through an active proxy. The proxy presents a very simple text-based messaging protocol to the Palm Pilot, but interacts with Sanctio using the more sophisticated AIM or ICQ procotols.

Because a user of the service may be reached on a number of different addresses (potentially one for each of the networks that Sanctio can communicate with), Sanctio must keep a large table of bindings between users and their current transport addresses on these networks. We used a distributed hash table DDS for this purpose.

9 Related Work

A number of projects share aspects of the Ninja vision of seamlessly interconnecting devices and Internet services. Related work can generally be characterized as addressing specific aspects of this problem space (such as supporting scalable services or embedding intelligence in the network), rather than taking Ninja's vertical approach to building a general-purpose Internet services platform. As the number of related projects in this domain is extremely large — spanning operating systems, programming languages, networks, embedded systems, and distributed computing platforms — we limit our discussion here to those projects which have taken a particularly complementary approach to the Ninja system design.

Flexible middleware systems, which support distributed computing across heterogeneous resources, are directly related to Ninja's goal of tying together Internet services with diverse small devices. CORBA [40] and DCOM [14] provide platform-independent, object-based network communication, although both systems are designed for tightly-coupled distributed applications and do not directly support composition and aggregation of components. Jini [39, 43] takes a Java-centric view, exploiting bytecode mobility to deliver stub code which implements a private communication protocol between client and service; stubs export a programming model based on Remote Method Invocation (RMI) [38]. Although Jini's literature describes a holistic distributed computing model not unlike that of Ninja, the system has been developed mainly for use within a workgroup, and does not provide security or scalability for the wide area. eSpeak [22] is another Java-based middleware system which intends to scale to the wide area, and to integrate PKI into its non-standard messaging layer. Neither system addresses service scalability and fault-tolerance, or access from impoverished devices which cannot run a Java-based communication protocol.

The goals of the Ninja Base environment are reflected by various application servers, including IBM WebSphere [24] and BEA Weblogic [4]. These systems strive to simplify the construction of scalable, fault-tolerant Internet services, generally requiring that applications be constructed as a set of Java components using an interface such as Enterprise Java Beans (EJB) [37]. EJB components are expected to be stateless or to manage their own state persistence; EJB components usually interact with a database to achieve the latter. vSpace differs from these application servers mainly by mandating an event-driven programming style (which facilitates high concurrency) and through the use of the DDS layer for persistence. The Ninja Base environment was inspired by earlier work on TACC [15] and SNS [9], both cluster-based Internet service platforms.

Harnessing intelligence in the network to transform and aggregate data across services has been investigated by several projects. Active networks [47] allow code to be injected into network routers to deploy new network protocols, implement traffic shaping, and perform packet filtering. An important distinction between these projects and Ninja's active proxies is the level at which data processing occurs; active networks operate at the transport or packet level, while active proxies operate using higher-level application semantics. As such, active proxies are not solely intended to implement protocols or perform packet-level operations; rather, they are used to perform service composition and aggregation, as well as soft-state transformations (such as HTML filtering, as demonstrated by the security proxy). While much of the work on mobile agents [28] has focused on supporting distributed artificial intelligence, active proxies share many of the same systems-level concerns, such as code mobility, naming, security, and coordination.

Many projects have used transcoding to adapt service content to better suit small devices [6, 15, 29, 34, 35, 36, 49, 50]. Additionally, a number of projects have attempted to develop universal interfaces for large classes of devices, including the recent WAP protocol stack [44]. Instead of assuming that a single standard will be adopted by all devices, the Ninja architecture allows multiple standards to be bridged by using active proxies as transformational intermediaries.

There are several additional technologies that we would like to explore as interesting examples of units. For example, Java Rings [11] and smart cards allow allow minimal computation, communication, and storage, but have no user interfaces. DIMM PC devices (matchbox-sized PCs on a single chip) could be used as mobile, computationally powerful devices that lack a user interface. Additionally, we believe that universal Plug and Play and Jini based devices could be easily integrated into the Ninja architecture.

10 Discussion and Future Directions

If Ninja succeeds in enabling connectivity between Internet services and diverse units, many new research directions will arise. The goal of moving intelligence into the network infrastructure, and opening up the infrastructure to allow anyone to push new components into it, raises questions about management, security, and service composition.

The first important concern is how to manage resources in a highly dynamic, decentralized network of active proxies. Operators should not be allowed to consume arbitrary amounts of network bandwidth, CPU, or memory; however, such restrictions cannot be made only on a per-site basis, as a given Operator may consume many aggregate resources across many active proxies. Otherwise, malicious operators could be used to launch distributed denial of service attacks against particular bases as well as the network itself. In the same vein, the infrastructure should prevent abuses of its content delivery mechanisms for unsolicited advertising or "spam" — already there are reports of people receiving unwanted advertisements via text paging to cellphones. If Ninja makes this problem worse, rather than better, the technology will not be adopted in the wide scale, or the infrastructure will remain closed.

New business models emerge in the world of ubiquitous network-based services. Today's model of funding websites through advertising revenue is inappropriate when services capture bits rather than eyeballs. Subscription and micropayment-based models are possible alternatives. In either case, retaining user privacy is an important concern as data and payments flow across the infrastructure. We envision a new "service marketplace" where both individual operators as well as entire vertically-integrated services are made available on a per-use or subscription basis. Another interesting model is that of a computational economy [31], where active proxies, services, and user agents participate in an automated marketplace where the commodities are CPU cycles, memory, and bandwidth. Service authors earn revenue by making their service available to others, and active proxies earn revenue by hosting services on behalf of users. Apart from the business implications, computational economies can be used to implement resource management, load balancing, and quality-of-service contracts.

Accessing powerful Internet services from small devices raises new challenges for user interface design. Ideally, service-to-device integration will be seamless. When failures do occur, however, the user may need some way to inspect or control the path of network components producing the fault. Exerting control over a network of active proxies from a device as limited as a text pager is difficult at best. Currently, networked devices are bound to a particular service; for example, a cellphone is used primarily for making phone calls. If the Ninja vision is realized, devices will become more versatile and the choices for using them more varied. Users will need some way to select between services and perhaps control a user profile used by those services.

Perhaps the largest challenge to face is that of automatically composing service components to meet the needs of particular devices. Expressing the transformation, caching, or aggregation properties of a Ninja operator in a type system is simple and potentially allows operators to be automatically chained into a path. However, the types must be expressive enough to capture the relevant semantics of an operator. For example, an English-to-French translation operator may take type `English text` as input, and `French text` as output; however, this alone does not imply translation between the two, as the operator might always output *"Je ne sais pas traduire cela texte."* Apart from strict type-matching, operator selection also depends upon consideration of an operator's quality, performance, and cost. Automatic path creation becomes a problem of balancing user requirements with other system demands, such as resource availability. Performing this operation efficiently and in a decentralized manner suggests several avenues for future research.

11 Conclusions

The Ninja architecture represents an important first step towards opening up the intrastructure of scalable, robust, adaptive Internet services. By opening the infrastructure, Ninja hopes to reclaim the distributed innovation that was responsible for the unprecedented success and widespread adoption of the Internet in the form of the world-wide web. Unlike the today's web, the service landscape envisioned by Ninja is one of active services and extremely diverse, mobile devices.

In this paper, we described the essential elements of this open architecture: robust service environments on clusters of workstations (bases), diverse devices (units), adaptive intermediaries to isolate services from units (active proxies), and a abstraction for the composition of these three elements (paths). In addition to describing our design and implementation of these components, we presented four innovative services that exploit the capabilities offered by this open infrastructure.

References

[1] America Online, Inc. The AOL Instant Messaging (AIM) Network. http://aim.aol.com/.

[2] 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*, pages 178–189, October 1998.

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

[4] BEA Systems. BEA WebLogic Application Servers. http://www.bea.com/products/weblogic/.

[5] Burton Bloom. Space/Time Tradeoffs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[6] Charles Brooks, Murray S. Mazer, Scott Meeks, and Jim Miller. Application-Specific Proxy Servers as HTTP Stream Transducers. In *Proceedings of the Fourth International World Wide Web Conference*, Dec 1995.

[7] P. Buonadonna, A. Geweke, and D. Culler. An Implementation and Analysis of the Virtual Interface Architecture. In *Proceedings of SC'98*, November 1998.

[8] Certicom. Elliptic Curve Cryptography for Palm VII. http://www.certicom.com/press/98/dec0298.htm, December 1998.

[9] Yatin Chawathe and Eric A. Brewer. System Support For Scalable and Fault Tolerant Internet Services. In *IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*, Lake District, UK, Sep 1998.

[10] 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.

[11] Dallas Semiconductor Designs. The Java Ring. http://www.ibutton.com/store/jringfacts.html.

[12] Datek Corporation. Datek Online Trading Service. http://www.datek.com, January 2000.

[13] UC Berkeley CS Division. The Millennium Project (home page), 1999. http://millennium.berkeley.edu.

[14] G. Eddon and H. Eddon. *Inside Distributed COM*. Microsoft Press, Redmond, WA, 1998.

[15] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-Based Scalable Network Services. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, St.-Malo, France, October 1997.

[16] Armando Fox and Steven D. Gribble. Security On the Move: Indirect Authentication Using Kerberos. In *Proc. Second International Conference on Wireless Networking and Mobile Computing (MobiCom '96)*, Rye, NY, November 1996.

[17] Armando Fox, Steven D. Gribble, Eric A. Brewer, and Elan Amir. Adapting to Network and Client Variability via On-Demand Dynamic Distillation. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, Cambridge MA, October 1996.

[18] I. Goldberg, S. D. Gribble, D. Wagner, and E. A. Brewer. The Ninja Jukebox. In *The 2nd USENIX Symposium on Internet Technologies and Systems*, Boulder, Colorado, USA, October 1999.

[19] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, Reading, MA, 1996.

[20] Steven D. Gribble, Eric A. Brewer, Joseph M. Hellerstein, and David Culler. Scalable, Distributed Data Structures for Internet Service Construction. In *Proceedings of the 4th USENIX Symposium on Operating System Design and Implementation (OSDI 2000)*, San Diego, California, USA, October 2000.

[21] Steven D. Gribble, Matt Welsh, Eric A. Brewer, and David Culler. The MultiSpace: an Evolutionary Platform for Infrastructural Services. In *Proceedings of the 1999 Usenix Annual Technical Conference*, Monterey, California, USA, Jun 1999.

[22] Hewlett Packard Inc. eSpeak: The Universal Language of E-Services. http://www.e-speak.net/.

[23] Jason Hill, Steven Ross, David Culler, and Anthony Joseph. A Security Architecture for the Post-PC World. Available at http://www.cs.berkeley.edu/~jhill/papers/SecPaper.ps.

[24] IBM Corporation. IBM WebSphere Application Server. http://www-4.ibm.com/software/webservers/.

[25] InfoWorld. Boeing to Put Net in the Air. http://www.infoworld.com/articles/hn/xml/00/04/27/000427enboeing.xml, April 2000.

[26] InfoWorld. E-cars take to the streets; wireless connections link road warriors to the Net. http://www.infoworld.com/articles/hn/xml/00/03/13/000313hnauto.xml, March 2000.

[27] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Mikio Takeuchi, Takeshi Ogasawara, Toshio Suganuma, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. Design, Implementation, and Evaluation of Optimizations in a Just-In-Time Compiler. In *Proceedings of the ACM 1999 Java Grande Conference*, June 1999.

[28] N. Jennings, K. Sycara, and M. Wooldridge. A Roadmap of Agent Research and Development. *Autonomous Agents and Multi-Agent Systems*, 1(1):7–38, July 1998.

[29] Liljeberg, M., et al. Enhanced Services for World Wide Web in Mobile WAN Environment. Technical Report C-1996-28, University of Helsinki CS Department, April 1996.

[30] S. Matsuoka, H. Ogawa, K. Shimura, Y. Kimura, K. Hotta, and H. Takagi. OpenJIT: A Reflective Java JIT Compiler. In *Proc. of OOPSLA '98, Workshop on Reflective Programming in C++ and Java*. http://openjit.is.titech.ac.jp/.

[31] Mark S. Miller and K. Eric Drexler. Markets and Computation: Agorics Open Systems. In Bernardo Huberman, editor, *The Ecology of Computation*. Elsevier Science Publishers/North Holland, 1998.

[32] Myricom Corporation. Myrinet: A Gigabit Per Second Local Area Network. In *IEEE Micro*, February 1995.

[33] PC World Communications, April 2000. http://www.pcworld.com/pcwtoday/article/0,1510,16045+1+0,00.html.

[34] Y. Sato. DeleGate Server, March 1994. `http://wall.etl.go.jp/delegate/`.

[35] M.A. Schickler, M.S. Mazer, and C. Brooks. Pan-Browser Support for Annotations and Other Meta-Information on the World Wide Web. In *Fifth International World Wide Web Conference (WWW-5)*, May 1996.

[36] B. Schilit and T. Bickmore. Digestor: Device-Independent Access to the World Wide Web. In *Proc. Sixth International World Wide Web Conference (WWW-6)*, Santa Clara, CA, April 1997.

[37] Sun Microsystems. Enterprise Java Beans Technology. `http://java.sun.com/products/ejb/`.

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

[39] Sun Microsystems. Jini Connection Technology. `http://www.sun.com/jini/`.

[40] The Object Management Group (OMG). The Common Object Request Broker Architecture. `http://www.corba.org`.

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

[42] J.R. von Behren, S. Czerwinski, A.D. Joseph, E.A. Brewer, and J. Kubiatowicz. NinjaMail: the Design of a High-Performance Clustered, Distributed E-mail System. In *Proceedings of the First International Workshop on Scalable Web Services*, Toronto, Canada, August 2000.

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

[44] WAP Forum. Wireless Application Protocol (WAP) Forum. `http://www.wapforum.org`.

[45] Matt Welsh, Nikita Borisov, Jason Hill, Rob von Behren, and Alec Woo. Querying Large Collections of Music for Similarity. Technical Report UCB/CSD-00-1096, U.C. Berkeley Computer Science Division, November 1999.

[46] Matt Welsh and David Culler. Jaguar: Enabling Efficient Communication and I/O in Java. *Concurrency: Practice and Experience*, 2000. Special Issue on Java for High-Performance Network Computing, To appear, `http://www.cs.berkeley.edu/~mdw/papers/jaguar-journal.ps.gz`.

[47] David J. Wetherall, John Guttag, and David L. Tennenhouse. ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols. In *Proceedings of IEEE OPENARCH'98*, San Francisco, CA, April 1998.

[48] Yahoo Finance. Yahoo Finance Investment Challenge. `http://contest.finance.yahoo.com/t1?u/`, 2000.

[49] Ka-Ping Yee. Shoduoka Mediator Service, 1995. `http://www.shoduoka.com`.

[50] Bruce Zenel and Dan Duchamp. A General Purpose Proxy Filtering Mechanism Applied to the Mobile Environment. In *Proceedings of the Third Annual ACM/IEEE Conference on Mobile Computing and Networking (Mobicom '97)*, New York, NY USA: ACM, 1997.

[51] Ben Y. Zhao and Anthony D. Joseph. XSet: A Lightweight Database for Internet Applications. Submitted for publication: `http://www.cs.berkeley.edu/~ravenben/publications/saint.pdf`, May 2000.