# Dependency Isolation for Thread-based Multi-tier Internet Services


Lingkun Chu* Kai Shen† Hong Tang* Tao Yang*,§ Jingyu Zhou*,§

* Ask Jeeves Inc.
Piscataway, NJ 08854
{lchu, htang}@ask.com

† Department of Computer Science
University of Rochester
kshen@cs.rochester.edu

§ Department of Computer Science
University of California at Santa Barbara
{tyang, jzhou}@cs.ucsb.edu



*Abstract*— **Multi-tier Internet service clusters often contain complex calling dependencies among service components spreading across cluster nodes. Without proper handling, partial failure or overload at one component can cause cascading performance degradation in the entire system. While dependency management may not present significant challenges for even-driven services (particularly in the context of staged event-driven architecture), there is a lack of system support for thread-based online services to achieve dependency isolation automatically. To this end, we propose *dependency capsule*, a new mechanism that supports automatic recognition of dependency states and per-dependency management for thread-based services. Our design employs a number of dependency capsules at each service node: one for each remote service component. Dependency capsules monitor and manage threads that block on supporting services and isolate their performance impact on the capsule host and the rest of the system. In addition to the failure and overload isolation, each capsule can also maintain dependency-specific feedback information to adjust control strategies for better availability and performance.**

**In our implementation, dependency capsules are transparent to application-level services and clustering middleware, which is achieved by intercepting dependency-induced system calls. Additionally, we employ two-level thread management so that only light-weight user-level threads block in dependency capsules. Using four applications on two different clustering middleware platforms, we demonstrate the effectiveness of dependency capsules in improving service availability and throughput during component failures and overload.**


## I. Introduction

Large-scale Internet service clusters are usually organized in multiple tiers, where external services are based on the aggregation of a large number of internal service components with multi-stage calling dependencies. At the single component level, event-driven service programming has been demonstrated to be highly efficient for constructing online services [1], [2], [3], [4]; however, such a style requires the partitioning of service programs into non-blocking event handlers. This is a challenging task for software development and maintenance, especially for progressively evolving services [5]. Thread-based systems provide a simpler programming model; but the weakness of multithreading lies in context switching overhead and poor caching performance at high concurrency levels. A common practice for managing highly concurrent Internet services is to impose a bound on the number of active threads [6], [7], [8], [9].

Component failures due to misconfiguration, lost networking connection, operational errors, and software bugs are common in large-scale Internet service clusters [10]. Together with server overload, these anomalies can result in slow-responding or unresponsive service components (faulty components). For component implementations based on a bounded-size thread pool, a slow-responding service component may block all threads at remote components that depend on it. This prevents the remote components from servicing their own requests and such an effect may spread to other service components through the dependency hierarchy.

This paper studies system availability problems aggravated by service calling dependencies. Our idea is to explicitly manage states of service threads in a per-dependency basis. We propose a new mechanism, called *dependency capsule*, which supports dependency isolation for service components implemented with thread-based programming model. A dependency capsule manages threads that block on calls to an internal device or to a remote component in the service cluster. More specifically, a capsule contains a thread pool and a management policy that decides whether a blocking service call should be delayed, terminated, or migrated to a different capsule (*e.g.*, migrated to the capsule handling connections to a replica of the originally-called remote component).

In addition to dependency isolation, dependency capsules also support quick bypass of faulty components when service semantics allow. It should be noted that dependency capsule is a technique orthogonal to replication. In particular, failure of a replica may not be detected promptly and the faulty replica could still cause cascading component failures when bounded-size thread pooling is used at service nodes. Dependency capsules can help quickly discover misbehaving components and bypass them.

To achieve automatic per-dependency management, service administrators specify a set of calling dependencies that need to be managed by dependency capsules and provide specific management policies for them. Different thread pool limits and management policies can be employed for different capsules.


This work was supported in part by NSF grants ACIR-0086061/0082666, EIA-0080134, CCF-0234346, CCR-0306473, and ITR/IIS-0312925, and by Ask Jeeves.

The proposed mechanism, *dependency capsule*, and its implementation have the following contributions: (1) Dependency-oriented thread state partitioning provides a per-component view of service dependencies, which helps to achieve dependency isolation. (2) Dependency capsules allow component-specific feedback for better fault tolerance and overload control. (3) Applications and clustering middleware can take advantage of dependency capsules without any change.

The rest of the paper is organized as follows. Section II describes the related work. Section III discusses the characteristics of multi-tier services and describes the problem statement. Section IV presents the architecture, design, and application interface of the dependency capsule. Section V illustrates implementation details. Section VI evaluates our approach using four applications on two different clustering middleware platforms, and Section VII concludes this paper.

## II. Related Work

Infrastructural software for cluster-based Internet services has been studied in TACC [11], MultiSpace [12], Ninja [13], and Neptune [14], [15]. These systems provide programming and runtime support on replication management, failure recovery and service differentiation. Our system can be incorporated into the above clustering middleware for supporting highly available thread-based services.

System availability is an important issue for large-scale cluster-based systems, which has been addressed extensively in the literature [16], [11], [12], [15], [13]. Typical metrics for measuring the overall system reliability are MTTR (mean time to failure) and MTTR (mean time to recovery). It often takes a long period of time to measure these metrics. Recently, the fault injection has been proposed as an effective but less time-consuming means to assess the system availability [17], [10].

Identifying blocked I/O routines and processing them with helper processes/threads are addressed in an earlier work of Flash Web server project [2]. SEDA divides applications into stages connected by events and a stage is executed by a bounded thread pool [3]. SEDA also studies the use of asynchronous IO to reduce the chance of thread blocking. A comparison of thread and event-driven programming was provided in [5] and the conclusion is that event-driven programming may provide better scalability; however it is often very difficult to maintain event-driven software in practice.

Recent Capriccio work by von Behren *et al.* argues that thread-based systems can achieve similar performance compared with event-based systems [18] through compile-time stack usage analysis and runtime optimization. While Capriccio can support hundreds of thousands of user-level threads, it may not work well under certain contexts (e.g., on SMP servers). Traditional kernel thread pool with a bounded size may continue to be used in practice. Our work focuses on improving availability of multi-tier services when the number of threads per node is bounded and inter-node dependency causes cascading performance degradation or failures.

Two-level thread management in dependency capsules is based on ideas from the previous work such as [19], [20]. Our management layer is built on top of the OS, targeting dedicated Internet service clusters with highly concurrent workload.

Resource accounting and scheduling are studied in [21], [22], [23], focusing on providing proportional resource allocation for different services or service classes. While our focus is to improve service availability and throughput during failure, their idea of resource usage separation influenced our dependency capsule design.

Goal-oriented programming analyzes component dependency specification to automatically extract parallelism [24]. Our goal is to improve service availability by managing component dependencies in dependency capsules.

## III. Background

An Internet service cluster hosts applications handling concurrent client requests on a set of machines connected by a high speed network. A number of earlier studies have addressed providing middleware-level support for service clustering, load balancing, and replication management [15], [11], [13]. In these systems, a service component can invoke RPC-like service calls or obtain communication channels to other components in the cluster. A complex Internet service cluster often has multiple tiers and service components depend on each other through service calls.

For example, Figure 1 shows the architecture of a three-tier search engine. This service contains five components: query handling frontends, result cache servers, tier-1 index servers, tier-2 index servers, and document servers. When a request arrives, one of the query frontends parses the request and then contacts the query caches to see if the result is already cached. If not, index servers are accessed to search for matching documents. Note that the index servers are divided into two tiers. Search is normally conducted in the first tier while the second tier database is searched only if the first tier index servers do not contain sufficient matching answers. Finally, the frontend contacts the document servers to fetch a summary for each relevant document. There can be multiple partitions for cache servers, index servers, and document servers. A frontend server needs to aggregate results from multiple partitions to complete the search.

Different component dependencies can be found in a multi-tier service cluster. We describe the following three classes of dependency relationships as examples. (1) *Replication dependency*. This category covers service invocation to replicas. For example, in Figure 1, the query handling frontends can call any of the document servers with replicated content. (2) *Bypassable dependency*. In this category, a next-tier service component can be bypassed without affecting the correctness of the service. For example, in Figure 1, the query handling frontends may bypass the cache server if needed. (3) *Aggregation dependency*. A service in this category accesses multiple supporting components and aggregates the results. For example, in Figure 1, a request requires an aggregation of results from multiple tier-2 partitions. Some applications

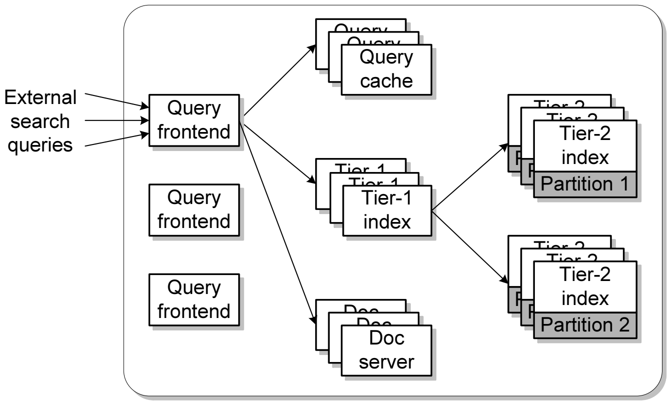


Fig. 1. A three-tier keyword-based document search service.

allow a subset of the supporting components to be used for service fulfillment though fewer components yield lower service quality [25], [26].

It is necessary to isolate these dependencies and manage them separately so that a failure in one component will not propagate to other components through the dependency hierarchy. In a thread-based system, bounded-size thread pooling is a common practice [6], [7], [8], [9]. When a component becomes unresponsive, all working threads of the calling component can gradually get blocked when waiting for the unresponsive component. Then the calling component may fail altogether when no working thread is available for servicing new requests. This effect can gradually propagate to other components in the service cluster. The problem can be somewhat mitigated with unbounded-size thread pooling, in which new threads are always created if all existing threads are blocked. Without bounding the thread pool size, a large number of threads can be accumulated during the failure of a supporting component. When such a failure recovers, blocked threads would simultaneously wake up and compete for resource, resulting in many requests timed out. The cause of the above unsatisfied results is that traditional thread programming model does not have a mechanism to recognize and manage these dependencies. To fill this gap, we propose dependency capsules to automatically recognize and manage these dependencies in thread-based programming model. Additionally, dependency capsules can also provide other features through per-dependency control such as feedback-based failure management.

## IV. Dependency Capsules

Motivated by the need of isolating the impact of unresponsive components, we propose a new thread management mechanism called *dependency capsule* that monitors and manages the blocking states of a thread based on service dependency. The basic idea is to consider the life cycle of a request at each node as going through a number of states in accessing network services offered within a cluster or local I/O devices. The main objectives of this work are summarized as follows.

**Dependency-aware concurrency management**. The traditional multithreading does not differentiate running threads and those that block on service calls to remote components or local devices. Our goal is to differentiate these states and handle them separately with appropriate concurrency management for each type of dependency.

**Automatic recognition and classification of resource dependencies**. The system should automatically recognize resource dependencies that may cause thread blocking. The system should also be able to automatically classify each blocking thread into a dependency capsule according to application-specified rules.

**Dependency-specific feedback for better availability and performance.** By identifying and managing service dependency, the system is able to maintain performance history for each type of dependency and provide this feedback information to application programs for service-specific control.

### A. The Architecture

We view each request handler at a service node as going through a set of states following service calling dependencies. The connectivity among these states is a star topology from the main working thread pool as illustrated in Figure 2. Each state is managed by a dependency capsule. A request is first processed by a user-level thread in the main working thread pool. Every time it performs a blocking operation, this thread enters the corresponding dependency capsule and returns back to the main thread pool at the completion of the blocking operation. Section IV-B discusses how our system assists application developers to specify necessary capsules through classification rules. Developers may choose not to specify dependency capsules for certain service invocation to reduce overhead.

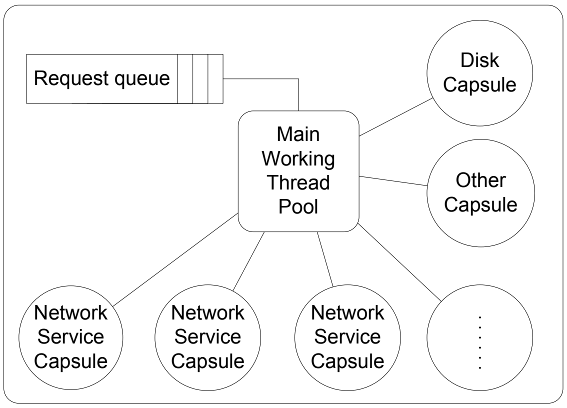


Fig. 2. Topology of dependency capsules at a service node.

A capsule is composed of the following four parts: (1) request handlers, (2) kernel threads, (3) scheduling policy, and (4) statistics. Request handlers directly run on user-level threads, which are scheduled on a pool of kernel threads. Each capsule can specify the upper bound on the kernel thread pool size, user-level thread pool size, the scheduling policy on how to prioritize user-level threads, and the timeout for the longest service waiting and processing time within this capsule. The timeout value can also be set according to the service level agreement to minimize the impact of false positives. The statistics module keeps track of resource consumption information. The statistics data are used by the capsule management and they are also provided as feedbacks to applications.

We briefly describe how a user-level thread (executing a service request) migrates among dependency capsules on a cluster node. Figure 3 shows the migration and state transition of a user-level thread between the main thread pool and a service capsule. When a user-level thread is created for processing an incoming request, it is first placed into the main working thread pool. All user-level threads in the main thread pool are scheduled by a set of kernel threads. If the user-level thread executes a blocking operation that should be managed in a dependency capsule, it will migrate to the appropriate capsule. Once the service completes, the corresponding user-level thread migrates back to the main thread pool. This procedure continues until the user-level thread completes the request processing.

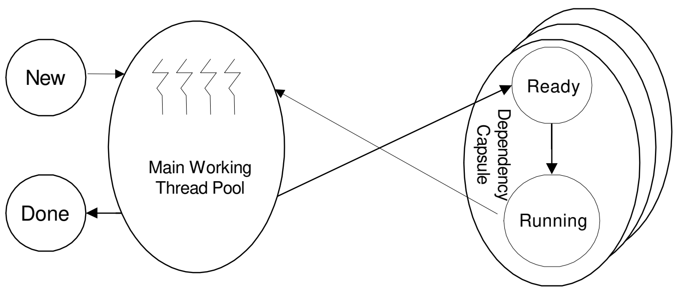


Fig. 3. State transition and capsule migration of a user-level thread.

There is a small cost for transferring a user-level thread from the main thread pool to a dependency capsule and then transferring back. The round-trip cost during code execution involves two kernel-level context switches. We do not transfer a user-level thread together with a kernel thread because we need to control the number of kernel threads in a capsule. Our experiment shows the migration cost is $40\mu s$ on a 450MHz PC and $16.5\mu s$ on a 2.4GHz PC. This cost is insignificant compared to the overhead of invoking a service on a remote service component.

### B. Capsule Types and Classification Rules

Currently we support two types of dependency capsules.

**Network service capsules.** A network capsule manages requests that block on network service calls to a particular remote component. Network communication such as remote procedure call or service connection, sending or receiving messages can be blocking. In our implementation, we intercept system calls that may result in blocking network operations and replace them with asynchronous calls. A monitoring thread watches on all sockets and checks if a user-level thread receives a network event. This mechanism takes advantage of the efficiency of event-driven style request processing in dealing with a large number of concurrent network service connections.

**Internal device capsules.** Service threads can also block on disk I/O or other internal device operations. We use helper threads in disk capsules to handle blocking I/O operations. The idea of using helper threads to provide asynchronous IO can also be found in Flash [2] and SEDA [3].

Each dependency capsule is associated with a set of classification rules, which determine what kind of operations the capsule handles so that a user-level thread can migrate to the capsule to carry out these operations. The capsule classification rules are listed as follows.

(1) A network service capsule can be classified based on the peer address. In addition to the support for typical Internet domain names or IP addresses, we also allow virtual service names to be specified as the peer addresses. A virtual service name will be dynamically translated into a real IP address when a service call is issued. We maintain a default network capsule to support network operations that do not match any specified classification rules.

(2) A disk capsule can be classified based on device names (with typical path wildcard support). For example, a classification rule "`/dev/sd[ab]`" will bind a disk capsule to two SCSI disks `/dev/sda/` and `/dev/sdb/`. All disk accesses to these devices will be handled by that capsule.

It is possible that an operation may be classified into multiple capsules. To prevent any ambiguity, we organize the capsules as a stack and an operation will be "trapped" by the top-most capsule that meets the following two criteria: (1) the operation's type matches the capsule's resource type; (2) the operation's parameters pass the capsule's classification rules.

### C. Use of Capsules in Load Throttling and Failure Management

Each network service capsule collects and maintains the performance statistics concerning the remote service component it handles. The performance statistics includes the number of blocking requests, their elapsed waiting time, and the recent average request response time. Capsule statistics can be used in the following ways.

**Caller-side Load Throttling.** Admission control is often employed on service components to handle overload. With the help of capsule statistics, admission control or load throttling can also be applied on the caller side. More specifically, we compare the number of blocking requests with a predefined threshold to determine if a request should be dropped at the caller-side.

Caller-side load throttling works better than server-side control in certain cases. For example, when a request needs to get service from multiple next-tier partitions, the server side admission control may make uncoordinated decisions on different partitions. This results in a waste of resource if the request cannot tolerate partition loss. In comparison, if a request drop decision is made at the caller side (*e.g.*, when a component overload is detected), next-tier partitions will not waste resource on this request. An additional benefit of caller-side load throttling is that it eliminates the cost of network communication and server-side management for requests that would be eventually dropped. This saving is especially important for fine-grain services that have relatively large request management overhead.

**Feedback-based Failure Management.** Another use of the capsule statistics is to provide feedback information to upper layer middleware or applications. For example, partition loss can often be tolerated with a degraded service quality in search applications [25]. Thus, when a partition encounters difficulty, service callers can bypass the problematic component to complete with degraded quality. Additionally, a component can be bypassed according to the service semantics. For instance, cache servers can often be skipped if they respond slowly. A service application can poll the number of queries accumulated in a cache capsule. If the cache server has too many queued requests, we can gradually decrease the timeout limit to phase out calls to the problematic cache server. This technique, we call *feedback-based timeout reduction*, protects the calling component from being affected by the unresponsive component while allowing it to quickly recover. We introduce a low watermark ($\alpha$) and a high watermark ($\beta$). Assume the original timeout value is $t$. When the number of accumulated queries $n$ is larger than $\alpha$, we calculate the actual timeout value as $\max\{0, t \times (\beta - n)/(\beta - \alpha)\}$. If this value is too small, the problematic component is bypassed. Our purpose of presenting this simple technique is to illustrate that dependency capsules can be utilized for failure management. More sophisticated techniques for quick detection of failures can be found in recent researches [27], [28].

### D. Application Interfaces

Application developers can define capsules through either configuration files or the capsule API interface. In either case, each capsule is uniquely identified by its name and resource category. Additionally, we can set various parameters, such as: (1) the number of kernel threads that are bound to the capsule, (2) the maximum number of user-level threads that can reside in the capsule, (3) the scheduling policy, and (4) the timeout value.

The maximum number of user-level threads of a capsule reflects the capacity of the next-tier component that the capsule depends on. It can be set to the same value as the maximum request queue length at the next-tier component. When the number of user-level threads exceeds the maximum value, an additional migration operation will fail and the application will be notified.

We provide several standard scheduling policies, such as FIFO and round-robin. Additionally, programmers have the flexibility to specify customized policies by providing a plug-in scheduling module (a dynamically linked library). The module defines several callback functions implementing the scheduling policy.

Configuration files provide a method for applications to specify capsules without recompilation. A capsule can be specified by its classification rules and various parameters mentioned earlier. For example, *destination = inet:192.168.1.0/24* specifies a network capsule that controls the communication to subnet 192.168.1.0/24. A configuration file can specify many dependency capsules. If there is no capsule corresponding to a certain type of resources, all operations belonging to that type will be executed in the main working thread pool.

In addition to configuration files, we also provide a set of functions that let an application dynamically create, monitor, control, and destroy capsules. This mechanism is necessary when machine names and capsule settings are not known a priori.

## V. Implementation

We have implemented dependency capsules on Linux Redhat 7.3 with GLIBC 2.2.5. Figure 4 depicts the architecture layout of the main system components. The capsule management component is responsible for the following tasks: (1) classifying threads into appropriate dependency capsules when they block; (2) migrating threads between dependency capsules and the main working thread pool according to the classification rules; and (3) maintaining performance statistics concerning the remote service component and providing caller-side load throttling and feedback-based failure management.

In addition to the capsule management, we also introduce a two-level thread management and a thread state transition capturing mechanism. Our two-level thread management, called *SBThreads*, allows threads migrate into and out of dependency capsules at the user-level. Through interposing system calls at the LIBC level and managing all network operations asynchronously, we can capture threads moving between blocking and non-blocking states without OS kernel support. Capturing such thread state transition is important for promptly trigger necessary capsule migrations. The rest of this section describes the implementation of the above three modules in detail.

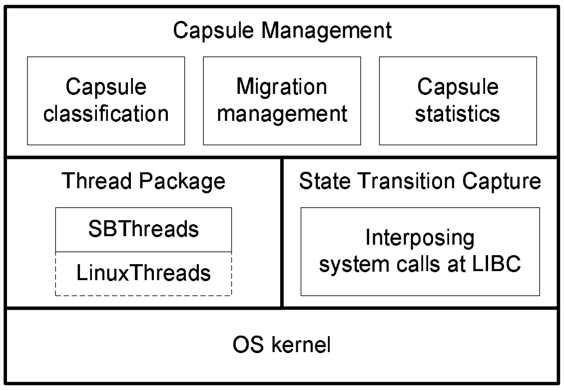


Fig. 4. Software layers of dependency capsule.

### A. SBThreads: A Two-level Thread Package

Our thread package uses an $m$-to-$n$ approach, *i.e.*, multiple user-level threads are scheduled by multiple kernel threads. The purpose of our user-level thread package is to provide explicit control over user-level threads (such as inter-capsule migrations) rather than to improve the performance. The following issues are worth noting.

*a) Interface Design.:* So far we have implemented 22 core POSIX thread functions for thread life control (creation, exit, and join), synchronization mechanisms (mutex and conditional variables), thread specific data, sleep and yield, and some other commonly used functions. We preserve the original semantics of these thread functions to achieve binary compatibility of existing programs.

*b) The Main Working Thread Pool.:* A request is first placed to a user-level thread and placed into the main working thread pool. The number of kernel threads in the pool should be properly configured. Too many working kernel threads can result in high context switch overhead and poor cache utilization. Typically application developers profile service modules to determine the bound based on the number of CPUs and application cache footprint. In terms of the scheduling policy, FIFO scheduling that follows the arrival order of user-level threads can be used to achieve fairness. Other policies (*e.g.*, giving a higher priority to threads that are in later stages of request handling [29]) can also be used.

### B. Inter-Capsule Migration Management

Each migration stops the current running user-level thread and places it to the destination capsule. The function involves the following steps: (1) find the current user-level thread; (2) save the current error number (`errno`) into the user-level thread control block; (3) set the stack frame of the current running kernel thread to that of the kernel scheduling thread; (4) switch to the kernel scheduling thread; (5) remember the stack pointer of the user-level thread; and (6) enqueue the user-level thread to the destination capsule.

### C. State Transition Capture

Inter-capsule thread migrations occur when threads move between blocking and non-blocking states, which are usually triggered by certain system calls. There are a number of ways to capture these system calls such as software wrappers, modifying kernel system call table, or dynamic interposition. Our approach exploits the fact that a system call always first goes through a wrapper function in LIBC. We modify the wrapper function in LIBC so that it will intercept the system call if needed. *The total number of lines modified in the GLIBC 2.25 source code is less than 30 lines*. Compared with kernel-based state transition capture approaches such as *scheduler activations* [19], our LIBC-based approach saves one kernel entrance for each interception.

When an application issues a system call such as a `read` or a `write`, we can tell whether it accesses a local device or a network object through the `fstat` facility. One drawback with this method is that it would incur too much overhead. Instead, we maintain a mapping table when a file handle is first created. Each entry in the table contains a pointer to the appropriate capsule. A file handle is treated as an index to the table. The actual handling of disk reads and writes is similar to the Flash Web server [2], where a number of kernel helper threads are dedicated for disk I/O operations.

Network system calls can be blocked indefinitely. If they are not carefully handled, it can lead to deadlocks when all kernel threads are blocked. To solve this problem, we first set all synchronous sockets as asynchronous. There is a master kernel thread in each network capsule to poll events on intercepted sockets. It also listens to a communication pipe to detect if there is a user-level thread entering the capsule. If there is an event, the corresponding user-level thread is set to be ready so a working kernel thread in the network capsule can execute them. The scheduling policy will be applied if several user-level threads are ready. If there is an incoming user-level thread, the master kernel thread registers the socket along with the user-level thread and continues its polling on the socket pool.

## VI. Evaluation

We experimented with our dependency capsule implementation on a Linux cluster. Services in our experiments are evaluated on two different clustering platforms for load balancing and replication management: (1) Neptune clustering middleware [15]; (2) JBoss J2EE-based platform [30]. Neptune, originating from an academic research prototype, is a clustering framework that supports service replication, location-transparent service invocation, failure detection and recovery, load balancing and resource management. It has been successfully used as the clustering middleware managing thousands of machines at the Web search engine sites Teoma and Ask Jeeves since 2000. JBoss is an open source implementation of the popular J2EE platform. It has 27% market share as a Java application server in IT production. For the JBoss platform, we provide dependency capsules through configuration files. No change is made in the JBoss platform. For Neptune, we support dependency capsules through both configuration files and API functions. In order to use the API functions, there is a small code change in Neptune so that the middleware load balancing and replication management does not interfere with our capsule management. There is no code change or recompilation needed for service applications except when applications desire explicit dependency-specific failure management as described in Section IV-C.

Our evaluation has the following objectives:

- Study the overhead of introducing dependency capsules in executing a network service application. In particular, we examine the migration overhead between the main thread pool and a service capsule.
- Demonstrate the improved availability by comparing the traditional multithreading with dependency capsules during a failure.
- Compare the availability and throughput of the traditional multithreading with dependency capsules when a failed component has replicas.
- Demonstrate the effectiveness of dependency-specific feedback in improving service availability and throughput.

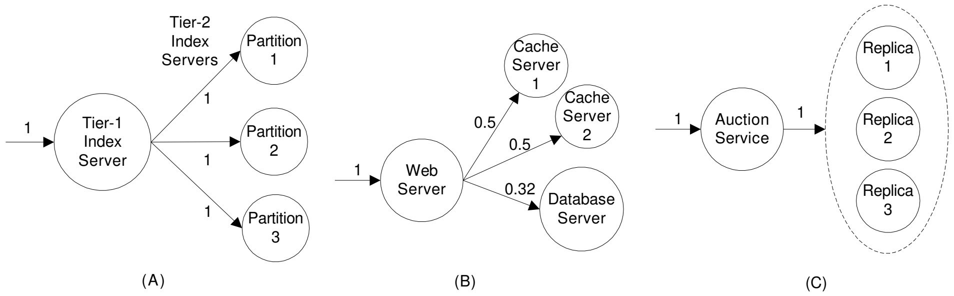


Fig. 5. Service dependency graphs for four benchmarks: (A) RET with Aggregation dependency; (B) BBS with Bypass-able dependency; (C) AUC and RUBiS with Replication dependency.

### *A. Evaluation Settings*

**Hardware.** Unless stated otherwise, experiments in this paper were conducted on a rack-mounted Linux cluster with 30 dual 400 MHz Pentium II nodes (each with 512 MB memory). Each node runs Redhat Linux with kernel version 2.4.18. All nodes are connected by a Lucent P550 Ethernet switch with 22 Gb/s backplane bandwidth.

**Application benchmarks and workloads.** We have used dependency capsules to support four applications.

1. *Search engine document retrieval service (RET).* The RET service emulates the document index searching component for a large-scale multi-tier keyword search engine. It scans through a number of document index partitions and returns an aggregated list of document identifications that are most relevant to the input query. In our experiment, indexed data are divided into three partitions. The service has a tier-1 index server which searches its own local database, and then collects additional results from three tier-2 index servers. The tier-1 index server sets a timeout value for each request. If the timer expires, it returns whatever it has collected from the three partitions. Since the result may not be complete, we introduce the notion of *quality-weighed throughput*, which indicates the throughput weighed by the quality of results. The quality weight of a result is defined as `number_of_partitions_returned` divided by `total_number_of_partitions`. When there is no partition loss, the weight is 1 and quality-weighed throughput becomes ordinary throughput. Figure 5(A) shows the service dependency graph of this prototype. The number on an edge in a service dependency graph indicates the probability of a request being forwarded from one service component to the other. In RET, all edges are labeled 1 because all three partitions will be searched for every request. Each partition is around 700 MB. The evaluation is driven by an uncached query trace obtained from the search engine *www.ask.com*. We proportionally adjust request arrival intervals to generate desired request demand.

2. *Bulletin board service (BBS).* The BBS service is an online discussion forum composed of four components: a Web server frontend, two partitioned document cache servers and a MySQL document database server. Requests in an online forum usually follow a Zipf distribution, *e.g.*, hot or recent topics tend to get more hits. Pages are equally distributed in these two cache servers based on their URL addresses. If there is a cache hit, the requested page is directly returned from the cache server. Otherwise, the page is generated from the database. Figure 5(B) shows the service dependency graph of this prototype. Following an earlier study [31], the cache hit ratio is set as 68%, thus a 32% accessing probability is labeled for the database server. Each query will be checked against cache. Since there are two cache partitions, the accessing probability for each cache server is 50%. We populated the document database with ten thousand articles before the experiment. We use a trace from the online discussion forum `www.melissavirus.com` dated on April 3, 1999.

3. *Online auction (AUC).* The AUC service is a prototype online auction service. Auction items are classified by a number of categories. The system allows users to list all items in one category, check the current bidding status of each item, place a bid on an item or add a new auction item for bidding. The service data that hosted in the MySQL database are replicated at three replicas. We use primary-secondary consistency for write operations [15]. Figure 5(C) shows the service dependency graph of this prototype. We use a synthetic trace described in [31] for this benchmark. About 5% requests involve writes. Item popularity, i.e. the number of bids each item receives, follows a Zipf distribution.

4. *Rice University Bidding System (RUBiS).* RUBiS is an auction service similar to AUC with the same dependency graph shown in Figure 5(C). Different versions of RUBiS are implemented using PHP, Java servlets and Enterprise Java Bean(EJB) [32]. We investigate the EJB version with bean-managed persistence(BMP) in this paper. The first tier is an Apache web server with a load balancing module, `mod_jk`, which distributes JSP and servlet requests to multiple Tomcat servers with session affinity. The second tier includes three replicated RUBiS servers running in JBoss All RUBiS servers contact a MySQL database server. The database is based on an SQL dump dated Jan. 23, 2004. We use the RUBiS client to produce a synthesized service workload.

The above four benchmarks represent three kinds of applications in terms of the component calling dependencies. The RET service represents the kind of services in which a request incurs an aggregation of results from multiple next-tier partitions (*Aggregation* dependency defined in Section III).

The BBS service belongs to the kind of services that allow some components to be be bypassed without affecting the service correctness (*Bypass-able* dependency). AUC and RUBiS represent services with replicated components (*Replication* dependency). These three types of applications are very common in Internet services. In our evaluation, we show dependency capsules can improve availability or performance for all these kinds of services.

**Availability evaluation setup.** Unless otherwise stated, we use a client request arrival rate representing about 70% of the backend server capability. For all tests, a request returned within two seconds is considered acceptable, or timeout otherwise (except that BBS uses 0.5 second as the timeout for its cache server). In addition to timeout, we apply an admission control policy so that no request will be directed to a server with at least 40 waiting requests.

**Abbreviations.** Here is a list of the abbreviations that we will use in the rest of this section: (1) **BTP** - traditional multithreading with a bounded-size thread pool (2) **UTP** - traditional multithreading with an unbounded-size thread pool (3) **DC** - basic dependency capsule mechanism (4) **DC+LT** - dependency capsules with load throttling (5) **DC+FB** - dependency capsules with feedback

## B. Overhead of Dependency Capsule

We investigate the overhead of dependency capsules, primarily on the cost of thread migration among capsules. We design a micro-benchmark to measure the migration cost from the main thread pool to a dependency capsule. This benchmark opens `/dev/null` and then writes one byte to it. We measure the time for each *write* operation. Note that a capsule migration involves two kernel thread context switches: one for migration from the main thread pool to the device capsule and the other for migration back. We measure the average cost for 100,000 migrations. Excluding the `write` system call cost, the round-trip migration cost is $40.1\mu s$ on a PIII-450MHz machine and $16.5\mu s$ on a PIV-2.4GHz machine.

Table I lists application benchmark performance executed using the traditional multithreading and dependency capsules. Introducing the dependency capsule results in a small increase (up to 5.8%) in response time, and little difference in terms of throughput.

| App. | Thrput (req/$s$) | | | RespTime ($ms$) | | |
|---|---|---|---|---|---|---|
| | BTP | DC | Overhead | BTP | DC | Overhead |
| RET | 38.20 | 38.76 | -1.5% | 136.6 | 141.3 | 3.5% |
| BBS | 68.87 | 68.87 | 0.0% | 58.7 | 61.4 | 4.6% |
| AUC | 33.82 | 34.14 | -1.0% | 144.2 | 151.9 | 5.3% |
| RUBiS | 79.96 | 79.90 | 0.1% | 36.3 | 38.4 | 5.8% |

TABLE I
APPLICATION PERFORMANCE WITH AND WITHOUT DEPENDENCY CAPSULES ON A PC CLUSTER.

Figure 6 shows the performance of the AUC benchmark with and without dependency capsules under different request arrival rates and the performance difference remains small.

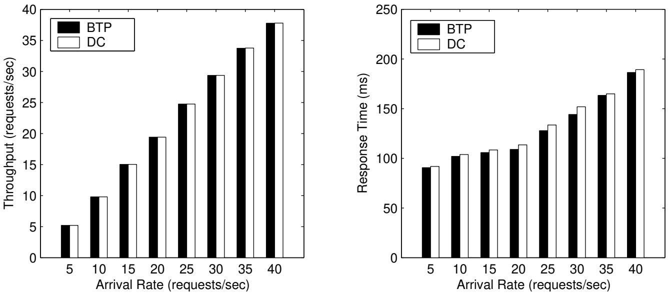


Fig. 6. AUC with/without dependency capsules under various request arrival rates.

## C. Effectiveness of Dependency Capsule for Improved Availability

In this experiment, we drive the RET service at 40 requests per second. We run the experiment for 30 seconds. At second 10, we inject a failure to partition 1 which then becomes unresponsive. It recovers at second 20.

Figure 7 shows the quality-weighed throughput (defined in Section VI-A) of three systems during the 30-second run time. When there is no failure, all systems work well. When partition 1 fails at second 10, the throughput of the system with bounded-size thread pooling quickly drops to zero because all threads in the thread pool of the aggregator server are blocked waiting for responses from partition 1. The problem is mitigated with unbounded-size thread pooling and dependency capsules. For the unbounded thread pool, new threads are created if all existing threads are blocked. Gradually, the aggregator accumulates a large number of threads. When partition 1 recovers, blocked threads wake up and simultaneously compete for CPU, resulting in many requests timed out. This is the reason that service throughput drops to almost zero shortly after partition 1 recovers. Besides this problem, unbounded thread pool can perform poorly under overloaded situation as demonstrated in SEDA [3].

For dependency capsules, threads blocked for contacting partition 1 are localized in a network service capsule while the main thread pool is not affected. During the failure, the throughput remains at about 2/3 of the throughput before failure because the quality weight becomes 2/3 after partition 1 fails. After partition 1 recovers, the contention is also avoided since the number of kernel threads remains the same in the main thread pool before and during the failure period. The system throughput quickly returns to a normal level. Notice that there is a throughput drop during the first a few seconds after partition 1 fails, this is because all requests wait for responses from partition 1 until the deadline expires.

Table II shows the average response time and loss ratio during the above experiment. The average response times are similar for all schemes since all requests are delayed until the timeout. However, the loss ratio is much smaller in DC. Replication, aggressive timeout, and admission control can be used if the service site demands higher availability during a failure.

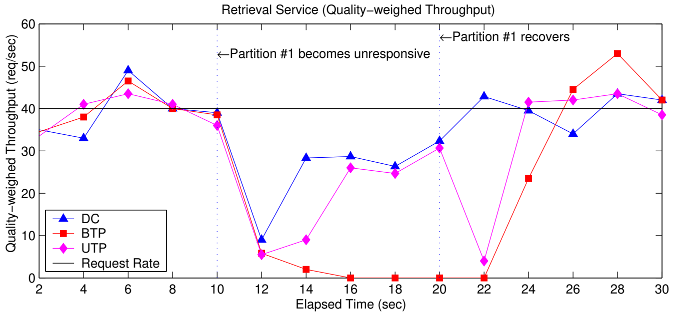


Fig. 7. Throughput of RET with multithreading or dependency capsules before/during/after a failure.

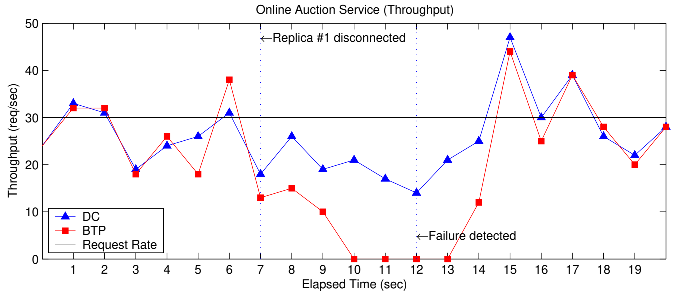


Fig. 8. Throughputs of AUC with and without dependency capsules under a lost network connection.

| Approach | Failure | RespTime ($ms$) | Thrput (req/$s$) | Loss Ratio |
|---|---|---|---|---|
| BTP | Failure | 280.6 | 39.5 | 0% |
| | Recovery | 1938.7 | 1.3 | 96.8% |
| UTP | Failure | 278.3 | 39.0 | 0% |
| | Recovery | 1452.0 | 16.6 | 58.5% |
| DC | Failure | 293.0 | 39.2 | 0% |
| | Recovery | 1488.3 | 27.9 | 30.3% |

TABLE II

AVERAGE RESPONSE TIMES AND LOSS RATIOS FOR RET.

| Approach | Failure | RespTime ($ms$) | Thrput (req/$s$) | Loss Ratio |
|---|---|---|---|---|
| BTP | Before | 145.3 | 25.1 | 0% |
| | Detection | 553.4 | 5.0 | 83.3% |
| DC | Before | 142.7 | 25.8 | 0% |
| | Detection | 241.3 | 19.4 | 35.3% |

TABLE III

RESPONSE TIMES AND LOSS RATIOS OF AUC UNDER A LOST NETWORK CONNECTION.

### *D. Availability with Component Replication*

In this experiment, we use the AUC service to evaluate the effectiveness of dependency capsules for improving service availability. We drive the system at 30 requests per second. As shown in Figure 8, at second 7, we disconnect one of the replicated back-end MySQL servers from the switch by taking away its Ethernet cable. The failure detection module of the Neptune clustering middleware running at the first tier server does not recognize this failure until five seconds later.

During this five second detection period, the bounded-size thread pooling gradually loses the capability to respond because threads are blocked gradually while requests are still sent to this disconnected machine. With the dependency capsule mechanism, the majority of requests (about 2/3) are processed smoothly during this five-second period since blocked threads are localized in the corresponding network service capsule. With two replicas, the system can still respond to most requests. The loss ratio shown in Table III also reflects this. Notice that there is a small percentage of requests involving writes; but the weak consistency model of the AUC service does not affect the throughput during this failure.

Additionally, we check the effectiveness of the dependency capsules in RUBiS when one of the replicated EJB server becomes unresponsive. We inject the failure by suspending the execution of an EJB server for 10 seconds. Figure 9 shows the throughput of RUBiS using the bounded-size thread pooling quickly drop to 0 during the failure. With the help of dependency capsules, 2/3 of the requests can still be served during the failure. Table IV shows the response time and loss ratio during the failure. The above experiment is done without any change in either the Apache server or the RUBiS source codes. The dependency capsules are activated using an external configuration file.

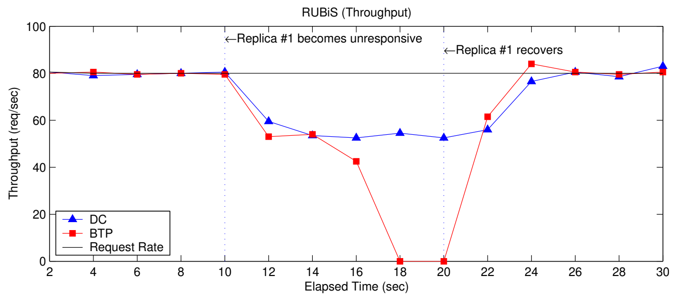


Fig. 9. Throughput of RUBiS (EJB version) with and without dependency capsules before/during/after a failure.

### *E. Effectiveness of Capsules in Load Throttling and Failure Management*

In this section, we evaluate the effectiveness of caller-side load throttling and feedback-based failure management using capsule statistics. We first test the effectiveness of applying caller-side load throttling based on runtime statistics collected from dependency capsules. We conduct the experiment on an aggregated retrieval service. In this experiment, we require that all partitioned data are available for each request to be successful, which is desirable for services that demand strong accuracy. We compare two schemes: (1) one with caller-side load throttling (DC+LT), and (2) the other with server-side admission control (DC). For both schemes, we fix the request arrival rate at 50 requests per second, which is roughly 125% of the system capacity. We run the experiments for 30 seconds. Figure 10 shows the result after the warm-up period. We can

| Approach | Failure | RespTime ($ms$) | Thrput (req/$s$) | Loss Ratio |
|---|---|---|---|---|
| BTP | Before | 29.4 | 80.3 | 0% |
| | During | 1291.8 | 29.9 | 62.6% |
| DC | Before | 28.9 | 79.8 | 0% |
| | During | 666.7 | 54.5 | 31.9% |

TABLE IV
RESPONSE TIMES AND LOSS RATIOS OF RUBIS BEFORE/DURING/AFTER A FAILURE.

see the scheme with caller-side load throttling performs much better than the other. It outperforms the server-side scheme by 40% on average. This is because each partition makes uncoordinated decision on which requests are to be dropped in the server-side scheme. When a request is dropped by one partition, other partitions may still process the request. Thus, resource is wasted when the application does not allow partition loss in the aggregated result. On the other hand, the caller-side load throttling can avoid this situation since the admission control happens on the client-side. It also avoids overhead in sending requests that will be eventually dropped on the server side.

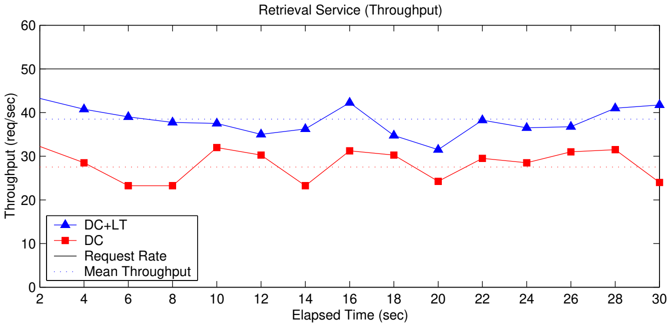


Fig. 10. Effectiveness of caller-side load throttling.

We evaluate the effectiveness of feedback-based failure management using the BBS service and the RET service.

**BBS service.** In this experiment, we examine the effectiveness of feedback mechanisms in improving service availability during a failure. We use the BBS service for this experiment. Request arrival rate of the system is set to be 70 requests per second. We run the experiment for 30 seconds. At second 10, cache server #1 becomes unresponsive. It recovers at second 20. We measure both the system throughput and response time for three schemes: (1) BTP; (2) DC, where we use 500 milliseconds as a timeout limit for the cache server; (3) DC+FB, where we apply the mechanism discussed in Section IV-C to contain the negative impact of a problematic cache server.

Figure 11 shows that the service throughput drops to almost zero for bounded-size thread pooling during the failure. It helps to a certain degree by setting a short timeout for cache (500 milliseconds). With dependency capsules, the system can deliver service for almost half of the requests that do not hit the failed cache server. For feedback-based failure management, the service is able to bypass the failed cache server and directly retrieves articles from the back-end database. Table V shows the average response times and loss ratios for three schemes. The average response time for feedback-based failure management is higher because a complete search operation is needed when the cache is not used.

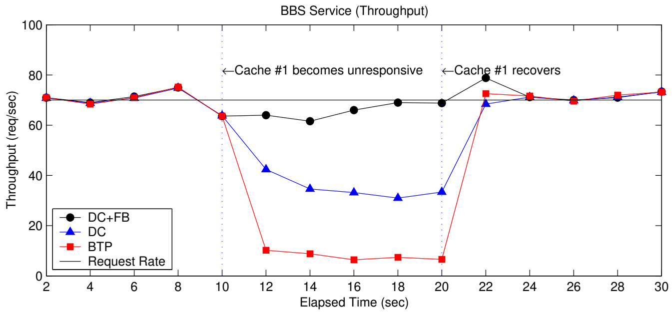


Fig. 11. Throughput of BBS using the traditional multithreading, dependency capsules without feedback or dependency capsules with feedback before/during/after a failure.

| Approach | Failure | RespTime ($ms$) | Thrput (req/$s$) | Loss Ratio |
|---|---|---|---|---|
| BTP | Before | 61.7 | 68.9 | 0% |
| | During | 965.0 | 8.0 | 87.8% |
| DC | Before | 60.2 | 68.9 | 0% |
| | During | 100.1 | 34.8 | 48.8% |
| DC+FB | Before | 61.8 | 68.9 | 0% |
| | During | 430.9 | 66.4 | 0.4% |

TABLE V
RESPONSE TIMES AND LOSS RATIOS FOR BBS.

**RET service.** In this experiment, we compare two schemes: the dependency capsule with feedback mechanism (DC+FB) and the basic dependency capsule mechanism (DC). DC+FB bypasses the partition 1 when the number of outstanding requests to the problematic partition exceeds a threshold. Figure 12 shows that throughputs for both schemes are similar. However when we compare the response time in Figure 13, we find the response time of DC+FB is much shorter than DC since subsequent requests in DC+FB do not need to contact the unresponsive partition when there are already a number of requests blocked on that partition. Thus, the feedback mechanism helps to reduce the response time of the RET service in this case.

## VII. CONCLUDING REMARKS

This paper presents the architecture, design, and evaluation of a new thread management mechanism called dependency capsule. Our primary goal is to improve the availability of multi-tier Internet services with complex component dependencies. Our design achieves three objectives: dependency-aware concurrency management, automatic recognition and classification of resource dependencies, and dependency-specific feedback for better availability. There is a small

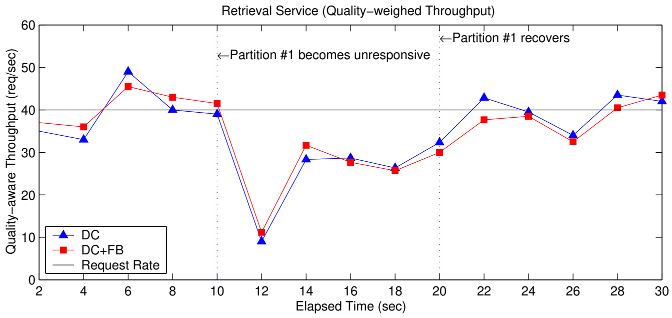


Fig. 12. Throughput of RET using dependency capsules without feedback and dependency capsules with feedback before/during/after a failure.

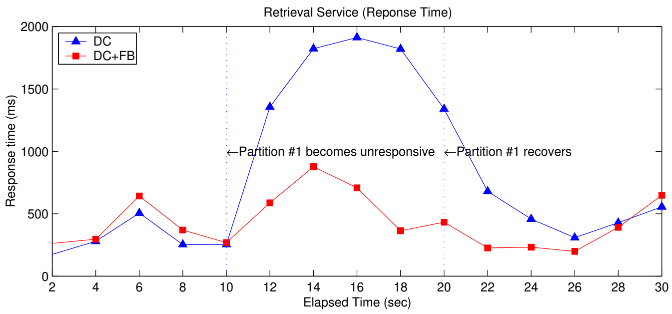


Fig. 13. Response time of RET using dependency capsules without feedback and dependency capsules with feedback before/during/after a failure.

overhead in executing network services with dependency capsules while the proposed techniques can greatly improve availability and throughput during component failures. We should emphasize that dependency isolation is still desirable at the presence of replication because the system may not be able to quickly identify the failure of a replica.

## ACKNOWLEDGMENT

We would like to thank the anonymous referees for their valuable comments and help.

## REFERENCES


[1] R. Morris, E. Kohler, J. Jannotti, and M. Frans Kaashoek, "The Click modular router," in *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, Kiawah Island, SC, December 1999, pp. 217–231.
[2] V. S. Pai, P. Druschel, and W. Zwaenepoel, "Flash: An Efficient and Portable Web Server," in *Proc. of 1999 Annual USENIX Technical Conf.*, Monterey, CA, June 1999.
[3] M. Welsh, D. Culler, and E. Brewer, "SEDA: An Architecture for Well-Conditioned, Scalable Internet Services," in *Proc. of the 18th ACM Symposium on Operating Systems Principles*, Banff, Canada, Oct. 2001.
[4] "Zeus Web Server," http://www.zeus.co.uk/products/ws/.
[5] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur, "Cooperative Task Management without Manual Stack Management," in *Proc. of 2002 USENIX Annual Technical Conf.*, 2002.
[6] "The Apache Web Server," http://www.apache.org.
[7] "BEA WebLogic," http://www.beasys.com/ products/ weblogic/.
[8] "IIS 5.0 Overview," http://www.microsoft.com/ windows2000/ techinfo/ howitworks/ iis/ iis5techoverview.asp.
[9] "Netscape Enterprise Server," http://home.netscape.com/ enterprise/ v3.6/index.html.
[10] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, "Why do Internet services fail, and what can be done about it?," in *Proc. of the 4th USENIX Symposium on Internet Technologies and Systems (USITS '03)*, Seattle, WA, Mar. 2003.
[11] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier, "Cluster-Based Scalable Network Services," in *Proc. of the 16th ACM Symposium on Operating System Principles*, Saint Malo, Oct. 1997.
[12] S. D. Gribble, M. Welsh, E. A. Brewer, and D. Culler, "The MultiSpace: An Evolutionary Platform for Infrastructural Services," in *Proc. of 1999 USENIX Annual Technical Conf.*, Monterey, CA, June 1999.
[13] J. Robert von Behren, E. A. Brewer, N. Borisov, M. Chen, M. Welsh, J. MacDonald, J. Lau, S. Gribble, and D. Culler, "Ninja: A Framework for Network Services," in *Proc. of 2002 Annual USENIX Technical Conf.*, Monterey, CA, June 2002.
[14] K. Shen, H. Tang, T. Yang, and L. Chu, "Integrated Resource Management for Cluster-based Internet Services," in *Proc. of the 5th USENIX Symposium on Operating Systems Design and Implementation*, Boston, MA, Dec. 2002.
[15] K. Shen, T. Yang, L. Chu, J. L. Holliday, D. A. Kuschner, and H. Zhu, "Neptune: Scalable Replication Management and Programming Support for Cluster-based Network Services," in *Proc. of 3rd USENIX Symposium on Internet Technologies and Systems*, San Francisco, CA, Mar. 2001.
[16] E. A. Brewer, "Lessons from Giant-Scale Services," *IEEE Internet Computing*, vol. 5, no. 4, pp. 46–55, 2001.
[17] K. Nagaraja, X. Li, B. Zhang, R. Bianchini, R. Martin, and T. D. Nguyen, "Using Fault Injection and Modeling to Evaluate the Performability of Cluster-Based Services," in *Proc. of the 4th USENIX Symposium on Internet Technologies and Systems (USITS '03)*, Seattle, WA, Mar. 2003.
[18] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer, "Capriccio: Scalable Threads for Internet Services," in *Proc. of the 19th ACM Symposium on Operating Systems Principles*, 2003.
[19] T.E. Anderson, B.N. Bershad, E.D. Lazowska, and H.M. Levey, "Scheduler Activations: Effective Kernel Support for User-level Management of Parallelism," *ACM Trans. on Computer Systems*, vol. 10, no. 1, pp. 53–79, Feb. 1992.
[20] K. K. Yue and D. J. Lilja, "Dynamic Processor Allocation with the Solaris Operating System," in *Proc. of the Intl. Parallel Processing Symposium*, Orlando, Florida, Apr. 1998.
[21] G. Banga, P. Druschel, and J. C. Mogul, "Resource Containers: A New Facility for Resource Management in Server Systems," in *Proc. of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, New Orleans, LA, Feb. 1999, pp. 45–58.
[22] D. G. Sullivan and M. I. Seltzer, "Isolation with Flexibility: A Resource Management Framework for Central Servers," in *Proc. of the 2000 USENIX Annual Technical Conf.*, San Diego, CA, June 2000.
[23] B. Verghese, A. Gupta, and M. Rosenblum, "Performance Isolation: Sharing and Isolation in Shared-Memory Multiprocessors," in *Proc. of the ACM 8th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 1998.
[24] R. van Renesse, "Goal-oriented programming, or composition using events, or threads considered harmful," in *8th ACM SIGOPS European workshop on Support for composing distributed applications*, Sintra, Portugal, Sept. 1998.
[25] L. Chu, H. Tang, T. Yang, and K. Shen, "Optimizing data aggregation for cluster-based Internet services," in *Proc. of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, California, June 2003.
[26] A. Fox and E. A. Brewer, "Harvest, Yield, and Scalable Tolerant Systems," in *Proc. of HotOS-VII*, Rio Rico, AZ, Mar. 1999.
[27] M. Y. Chen, A. Accardi, E. Kiciman, D. Patterson, A. Fox, and E. Brewer, "Path-Based Failure and Evolution Management," in *Proc. of the 1st Symposium on Networked Systems Design and Implementation.*, San Francisco, CA, Mar. 2004.
[28] M. Welsh and D. Culler, "Adaptive Overload Control for Busy Internet Servers," in *Proc. of the 4th USENIX Symposium on Internet Technologies and Systems (USITS '03)*, Seattle, WA, Mar. 2003.
[29] J. C. Mogul and K. K. Ramakrishnan, "Eliminating Receive Livelock in an Interrupt-driven Kernel," in *Proc. of USENIX Annual Technical Conf.*, San Diego, CA, Jan. 1996.
[30] "JBoss," http://www.jboss.org/.
[31] H. Zhu and T. Yang, "Class-based Cache Management for Dynamic

Web Contents," in *Proc. of IEEE INFOCOM'2001*, Anchorage, AK, Apr. 2001.

[32] E. Cecchet, J. Marguerite, and W. Zwaenepoel, "Performance and Scalability of EJB Applications," in *Proc. of the 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications.*, Seattle, WA, Nov. 2002.