Dependence Isolation for Highly-Available Multi-tier Service Clusters

The Authors

Tao Yang

- BS in CS and ME in AI from Zhejiang Univeristy, China – 1984, 1987
- MS and PhD in CS from Rutgers – 1990, 1993
- Full professor at University of California at Santa Barbara
- Chief Scientist and VP of R&D for Teoma Tech., a startup Internet Search in 2001
- Currenly VP and Chief Scientist for Teoma Tech at Ask Jeeves

The Authors

Kai Shen

- Received PhD in CS from U of Cal at Santa Barbara in 2002
- Joined faculty of CS Dept. at U of Rochester that year as an assistant professor

Jingyu Zhou

Whereabouts:

UNKNOWN

Activities:

UNKNOWN

Academic Career:

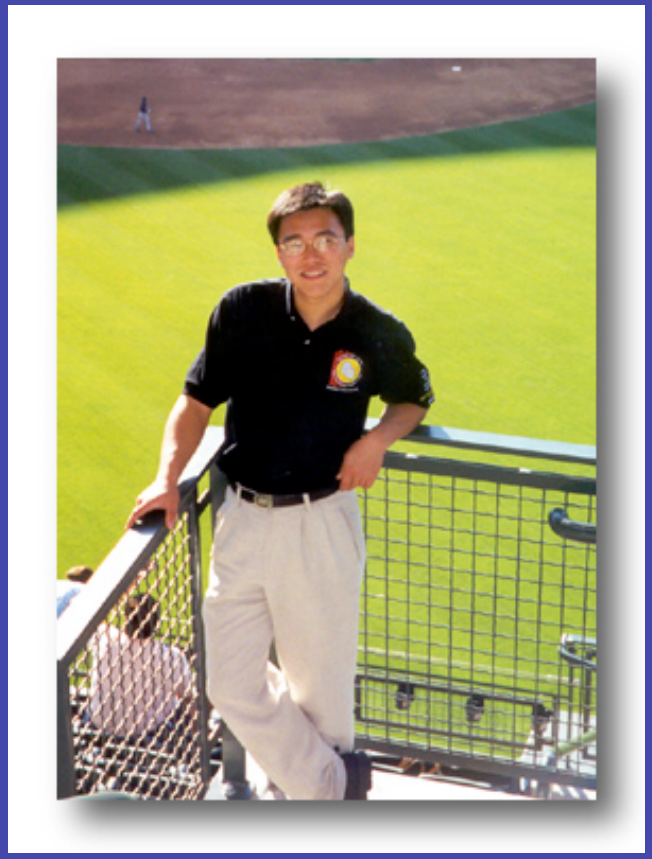
UNKNOWN

Was at least a student at U of Cal at Santa Barbara at some point…

The Authors

Lingkun Chu

- PhD student at U of Cal at Santa Barbara
- interested in cluster-based network services



Hong Tang

- PhD student at U of Cal at Santa Barbara
- Interested in middleware support for large apps running over WANs

Introduction

The Problem:

- In multi-tier web services' interdependencies can make a single blocking thread a performance disaster
- even in web services with only a small amount of interdependence, overload can cause the same lock-ups
- even if web services are free of overload, faulty components can still cause service lock-ups

WHAT CAN WE DO?

Introduction

The Solution:

Capsules are your friends!

Introduction

What do capsules do?

- provide *dependence isolation*
  - if a thread blocks, the capsule will keep it from gumming up the whole works
  - if a module is faulty, the capsule will isolate and bypass it
  - if a module is overloaded, the capsule will release some of the pressure on it
  - provide valuable statistics…

Background

Current system is *monolithic multithreading:*

- impose a limit on amount of concurrent threads running in order to reduce context switching overhead and poor cache performance

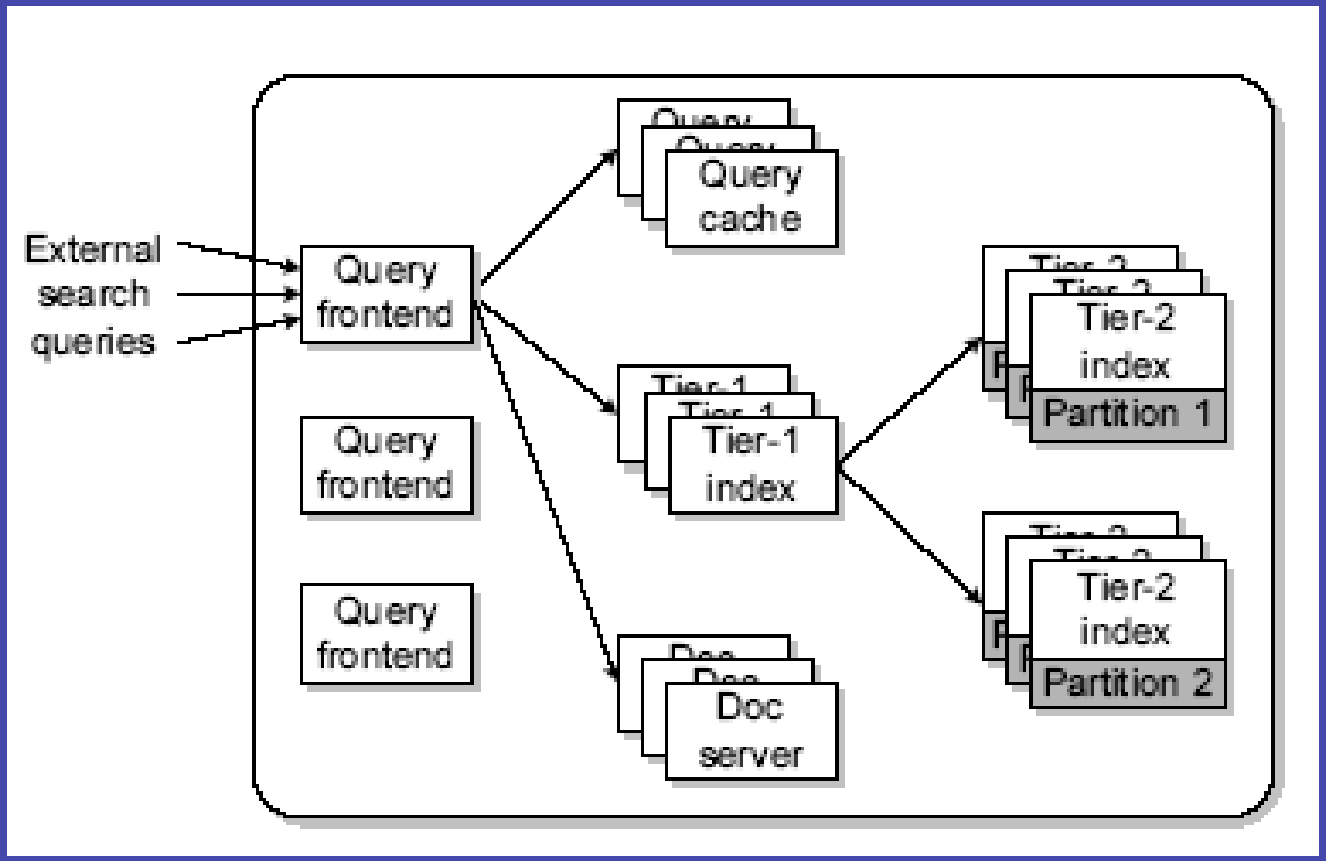
However…

One slow-responding node or service can hold up all the threads waiting for it

This can be solved by removing the upper bound to allow more threads to use the slow service / resource, but then performance is degraded for the reasons above.

Background

Example of composed services



Background

Is this really insurmountable with monolithic multithreading?

Apparently… Yes…

- Setting the thread pool bound lower makes less use of resources

- Setting the thread pool bound higher performs better only to a point, since poor use of system resources (cache and context switches) dominates execution time

Why?

"…it does not differentiate blocked threads from running threads."

- All threads are treated the same, whether they should be or not

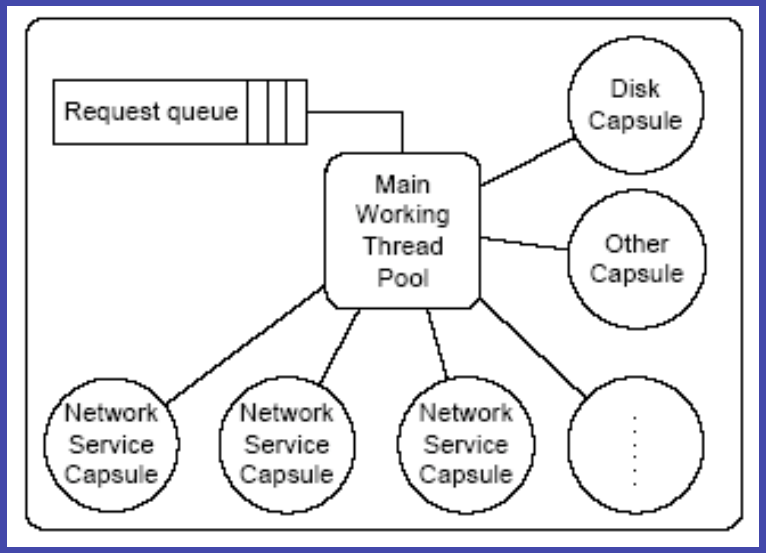
- This is the main weakness of monolithic multithreading

Dependence Isolation

What does it do?

- Dependence-aware concurrency control:
  - manage blocking threads and running threads differently
- App. management with app-specific dependence control
  - let system recognize resource dependency
  - allow developers to specify type of dependency that *should* be recognized
- Dependence-specific statistics
  - maintain performance history for each dependence type
  - provide feedback to applications for service-specific control

Architecture



- Requests are queued and processed at the Main Thread Pool
- If a Thread in the Main Pool executes a blocking operation it is moved to the appropriate capsule
- If a thread blocks for too long the caller can try a different node or bypass the service, if allowed
- When the Thread's call finishes it is moved back into the Main Pool to continue executing

Each capsule consists of:

1) user-level threads
2) kernel-level threads
3) a scheduling policy
4) statistics

The Plot Thickens

Possible pitfall:

- With all this switching back and forth of threads, why isn't context switching a bottleneck as it would be for monolithic multithreading?

- It all goes really fast, 40_s on a 450 MHz, and 16.5_s on a 2.4 GHz

- Apparently this cost is negligible when compared with the overhead of an actual service invocation

- Threads are only migrated on blocking calls… so if the service has no / few blocking calls, context switching won't come up often

Services

• Reading statistics provided by the API can allow client-side load throttling

- If a client (or client's program) knows an application needs to use a large set of interdependent server modules, it isn't worth running until all are free

- Server-side admission control has no notion of what requests are being made together

- This allows client to make decisions about when to submit its requests, taking some burden off of the server

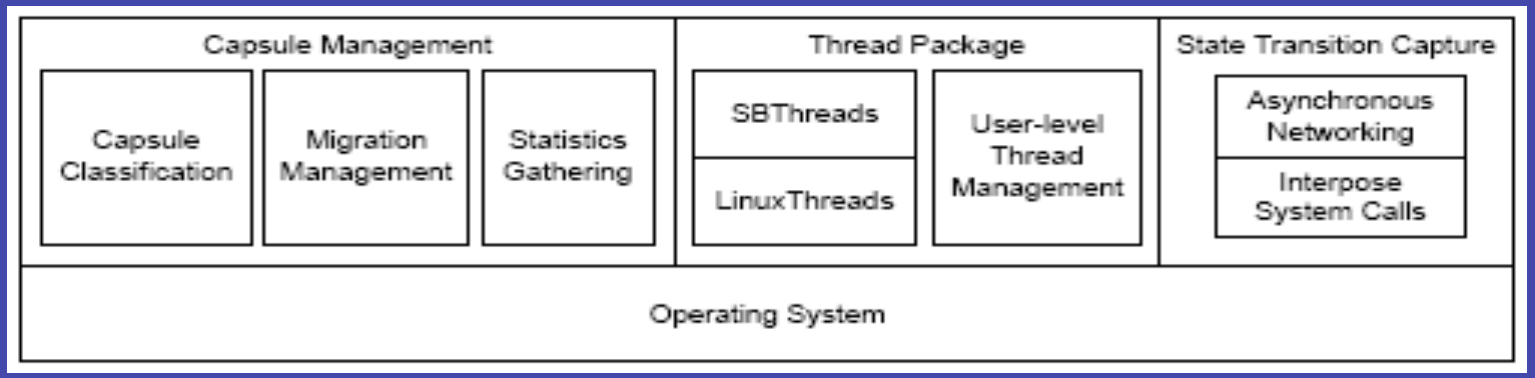
Implementation

Time to settle in, folks…

- In order to implement their *SB Threads* the authors needed to capture threads as they blocked and unblocked.

- Using the OS' kernel support would be a half-solution, since it would seriously specialize the code

- Found that by interposing system calls at the LibC level and managing all network calls asynchronously, threads can be captured on state change



Implementation

Capturing Threads

• Authors exploited the fact that system calls inside a program need to pass though the GLibC wrapper

• It was apparently a simple thing to modify the GLibC wrapper to intercept or pass through system calls… less than 30 lines of code were added / modified

• Mapping to capsule types is handled by a table that maps file descriptors to capsule types

• So far 15 system calls are supported:

IO Operations:

open *close* *pread* *pwrite* *read* *write* *fsync*

Socket Operations:

accept *connect* *recv* *send* *sendto* *recvfrom* *recvmsg* *sendmsg*

Implementation

Timing

It isn't just enough to create User threads and execute them in dependence capsules, the Kernel threads that underlie the User threads need to be protected.

If a User thread makes a call to timing routines, such as usleep, it will block the Kernel thread too…

Solution: Authors added another Kernel thread to specifically handle such calls and rerouted all calls of Kernel-thread-blocking routines to it

When a User thread calls such a routine, this special Kernel thread will handle it, changing the User thread's state and putting it to sleep

Later the special Kernel thread will wake up when the User thread would have and will put the User thread back into its *dependence capsule* to continue executing

Implementation

Inter-Capsule Migration

- If a thread is happily executing in one capsule and issues a blocking call to a resource governed by a different capsule, it needs to be managed by *that* other capsule until the call runs its course.

There *is* still a cost for migrating the threads around between capsules, so it is necessary to be certain that a specific call will actually block.

In order to avoid an unnecessary user and kernel context switch, the call is first tried in the Main Thread Pool… if it actually would block, the thread is transferred to the appropriate capsule, otherwise the call continues to be executed in the Main Thread Pool

Implementation

Feedback-based Failure Management

Remember those capsule statistics? Now they earn their pay…

- **Polling:** an app. can actively get the status of a capsule
- **Event-triggered:** an app. can also register callback functions and their conditions in a dependence capsule

Now not only can clients find out exactly how well the server is bearing up, but the service's developer can have it act on its own when certain conditions come up.

This is a stronger version of server reliability than the actual dependence isolation, since the behavior of the service can be made to change based on the condition and capability of its resources, represented by the dependence capsules.

Evaluation

Hardware:

Rack-mounted Linux cluster consisting of:

- 30 dual 400 MHz Pentium II nodes
- 512 MB RAM on each
- each running Redhat Linux v2.4.18
- all nodes connected via a Lucent P550 Ethernet switch with bandwidth of 22 Gbps

Evaluation

Benchmarks:

- **Search Engine Document Retriever (RET)**
  - Consists of multiple tiers of indexes
  - Uppermost tier sets timeout for collecting results from lower tiers

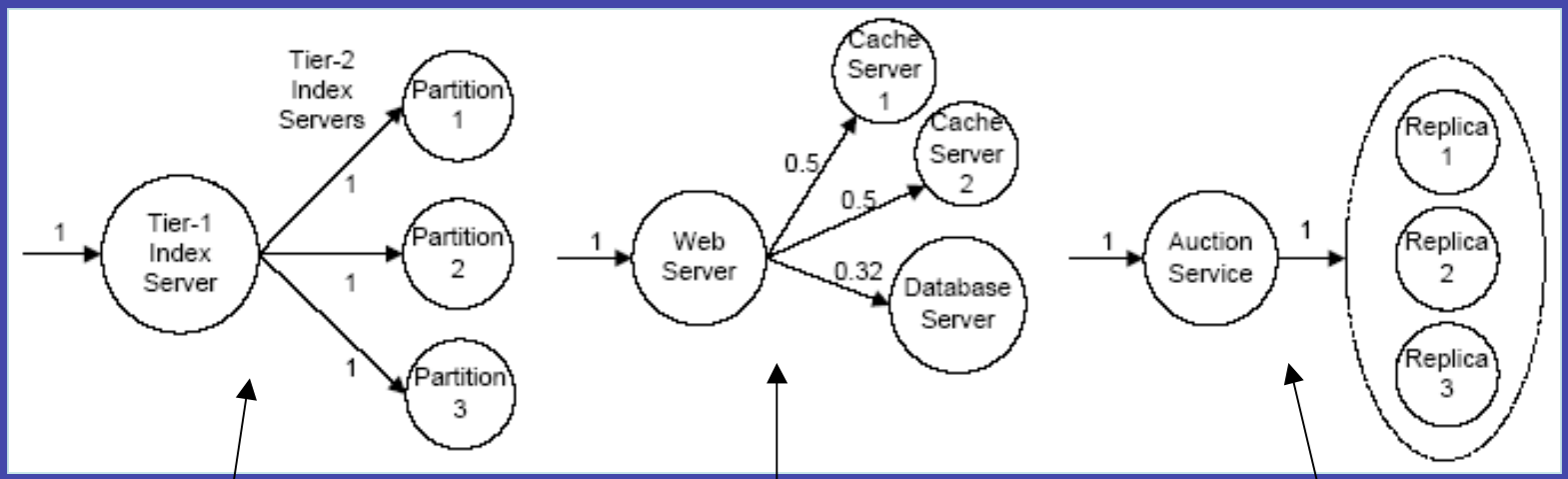
  Metric: quality-aware throughput = (# of indexes returned) / (# of indexes)
- **Bulletin Board Service (BBS)**
  - Consists of multiple tiers that intercommunicate
  - Cache server is hit before lower document servers

  Metric: Throughput
- **Auction Service (AUC)**
  - Consists of multiple replicas of the auction item database

  Metric: Throughput

Evaluation

Dependence Graph for Test Services



RET – Search engine

BBS – Bulletin board

AUC – Auction server

Evaluation

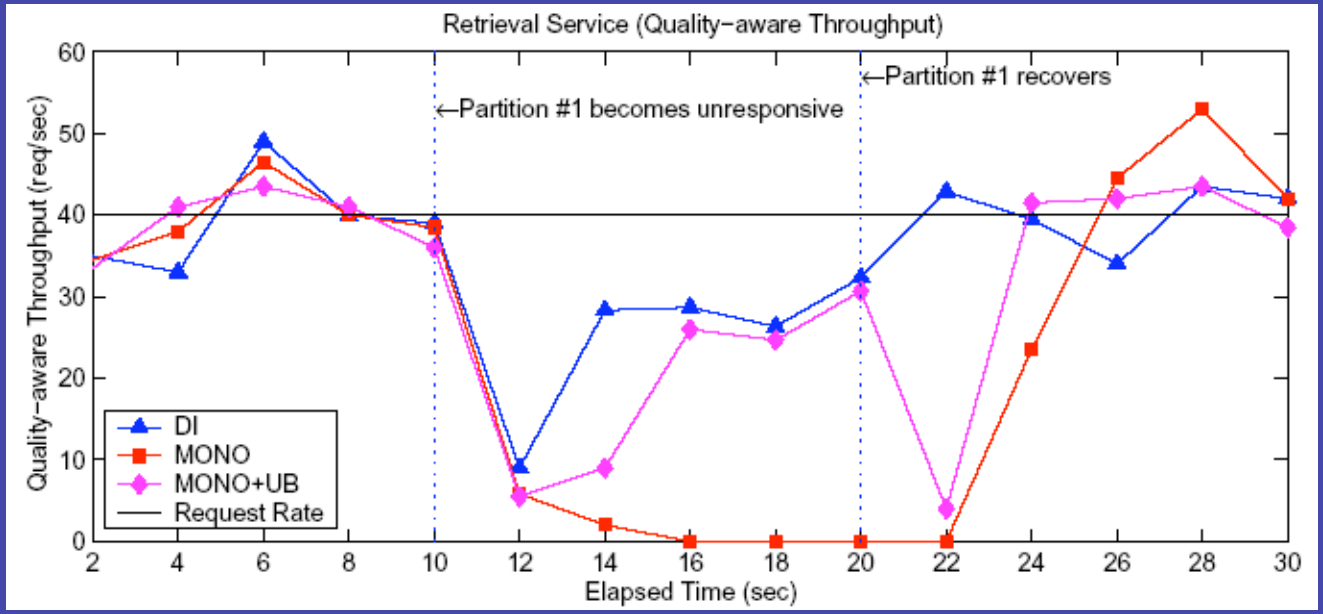
Availability setup:

- Sustain client request arrival rate at 70% of backend server's capacity
- Request for service returned ≤ 2 seconds is considered acceptable, otherwise is considered a timeout (BBS uses 0.5 seconds as its timeout)
- Admission control through waiting request queue length, with a maximum of 40 waiting requests
- Requests sent when queue is full are rejected and *are counted as failed requests*

Faults:

- Server is just really slow, but still is alive – fault injected into app. software
- Server croaks entirely – network cable is removed

Evaluation



RET service

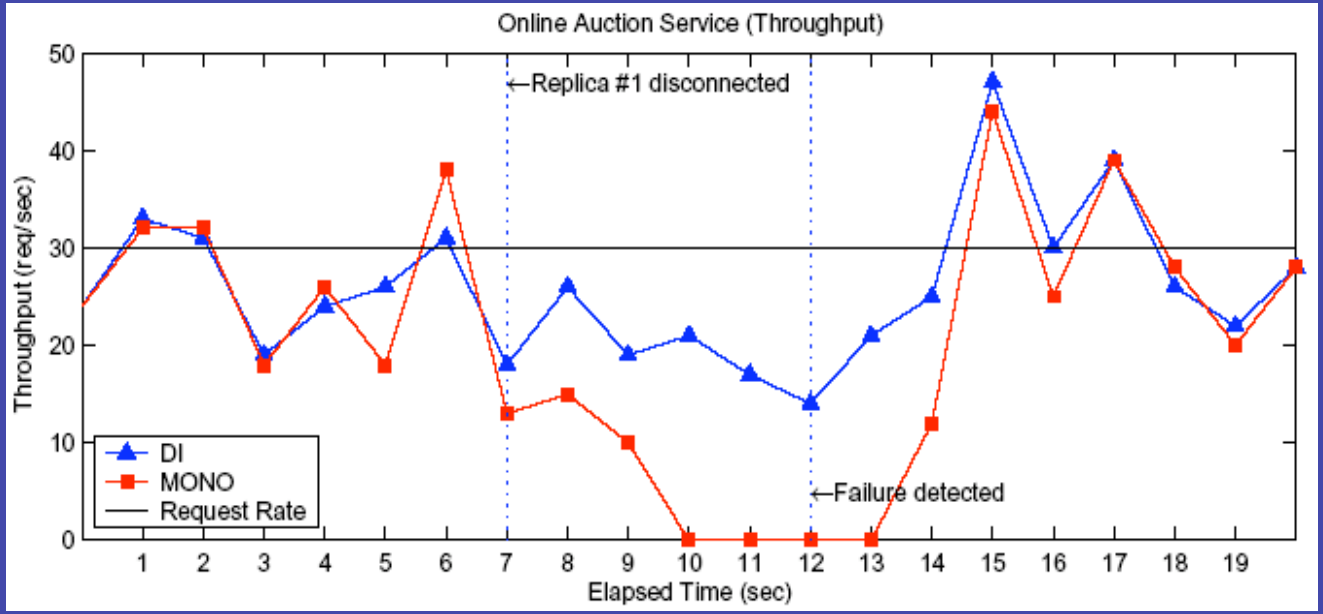
- 40 req per sec
- fail at 10 sec
- recover at 20

DI = dependence isolation

MONO = monolithic multithreading

MONO+UB = monolithic multithreading with dynamic upper bound

Evaluation



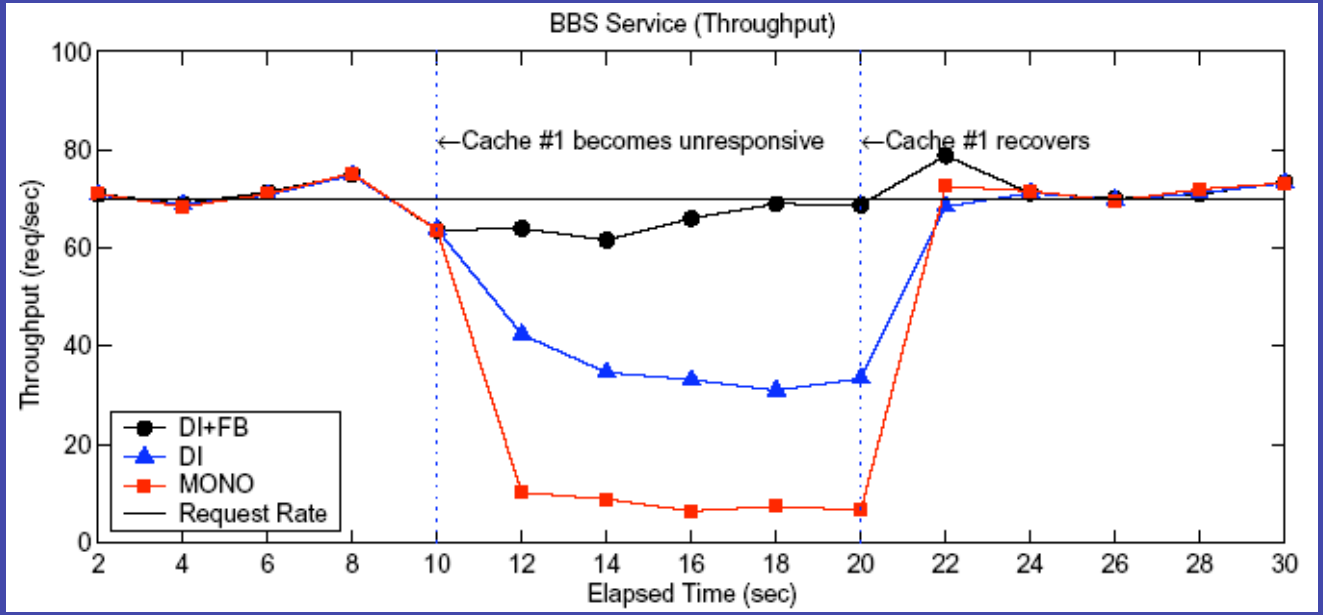
AUC service

- 30 req per sec
- fail at 7 sec
- fail found at 12

DI = dependence isolation

MONO = monolithic multithreading

Evaluation



BBS service

- 70 req per sec
- fail at 10 sec
- recover at 20

DI + FB = dependence isolation and feedback-based failure management

DI = dependence isolation

MONO = monolithic multithreading

Conclusions

Dependence isolation capsules are good!

They do all they promise to:

- Threads that block and threads that run can be treated differently
- Feedback-based Failure Management is a powerful tool that can keep a service running well even if parts are overloaded or fail

Wake Up Call

It’s now safe to wake up and / or remove your earplugs!