CS514: Intermediate Course in Operating Systems

Professor Ken Birman
Vivek Vishnumurthy: TA

Programming Web Services

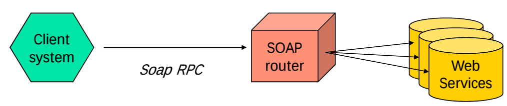
- We've been somewhat client centric
  - Looked at how a client binds to and invokes a Web Service
  - Discussed the underlying RPC protocols
  - Explored issues associated with discovery
- But we've only touched upon the data center side
- Today discuss the options and identify some tough technical challenges

(Sidebar)

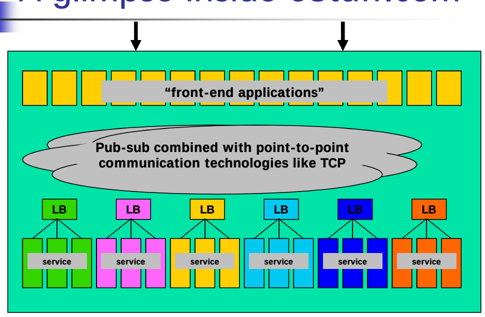
- Not all Web Services will be data centers
  - Intel is using Web Services to access hardware instrumentation
  - Many kinds of sensors and actuators will use Web Services interfaces too
  - Even device drivers and other OS internals are heading this way!
- But data centers will be a BIG deal…

Reminder: Client to eStuff.com

- We think of remote method invocation and Web Services as a simple chain
- This oversimplifies challenge of "naming and discovery"



A glimpse inside eStuff.com



What other issues arise?

- How does one build scalable, cluster-style services to run inside a cluster
  - The identical issues arise with CORBA
- What tools currently exist within Web Services?
- Today: explore process of slowing scaling up a service to handle heavier and heavier loads
  - Start by exploring single-server issues
  - Then move to clustering, and role of the publish-subscribe paradigm
  - We'll touch on some related reliability issues

Building a Web Service: Step 1

- Most applications start as a single program that uses CORBA or Web Services
  - Like the temperature service
  - Exports its interfaces (WSDL, UDDI)
  - Clients discover service, important interfaces and can do invocations

Suppose that demand grows?

- Step 2 is to just build a faster server
  - Port code to run on a high-end machine
  - Use multi-threading to increase internal capacity
- What are threads?
  - Concept most people were exposed to in CS414, but we'll review very briefly

Threads

- We think of a program as having a sort of virtual CPU dedicated to it
  - So your program has a "PC" telling what instruction to execute next, a stack, its own registers, etc
- Idea of threads is to have multiple virtual CPUs dedicated to a single program, sharing memory

Threads

- Each thread has:
  - Its own stack (bounded maximum size)
  - A function that was called when it started (like "main" in the old single-threaded style)
  - Its own registers and PC
- Threads share global variables and memory
- The system provides synchronization mechanisms, like locks, so that threads can avoid stepping on one-another

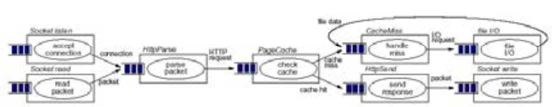
Challenges of using threads

- Two major ways to exploit threads in Web Services and similar servers

1. Each incoming request can result in the launch of a new thread
2. Incoming requests can go into "request queues". Small pools of threads handle each pool

- We refer to these as "event" systems

Example Event System



(Not limited to data centers… also common in telecommunications, where it's called "workflow programming")

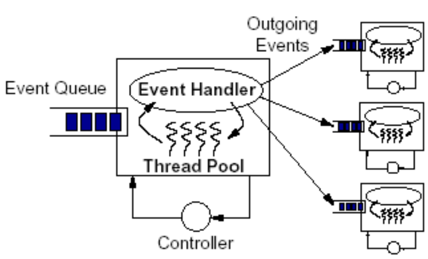
Problems with threads

- Event systems may process LOTS of events
- But existing operating systems handle large numbers of threads poorly
  - A major issue is the virtual memory consumption of all those stacks
  - With many threads, a server will start to thrash even if the "actual workload" is relatively light
  - If threads can block (due to locks) this is especially serious
- See: Using Threads in Interactive Systems: A Case Study (Hauser et al; SOSP 1993)

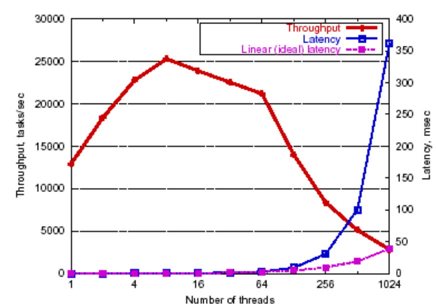
Sometimes we can do better

- SEDA: An Architecture for Well-Conditioned, Scalable Internet Services (Welsh, 2001)
  - Analyzes threads vs event-based systems, finds problems with both
  - Suggests trade-off: stage-driven architecture
  - Evaluated for two applications
    - Easy to program and performs well

SEDA Stage

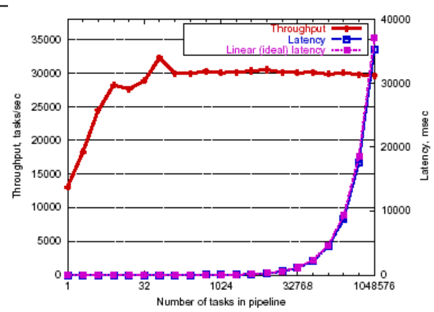


Threaded Server Throughput



Source: SEDA: An Architecture for Well-Conditioned, Scalable Internet Services (Welsh, SOSP 2001)

Event-driven Server Throughput

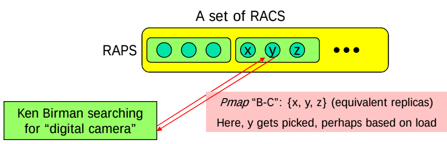


What if load is *still* too high?

- The trend towards clustered architectures arises because no single-machine solution is really adequate
- Better scheme is to partition the work between a set of inexpensive computers
  - Called a "blade" architecture
  - Ideally we simply subdivide the "database" into disjoint portions

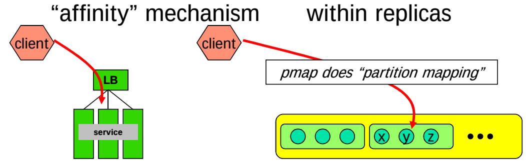
A RAPS of RACS (Jim Gray)

- RAPS: A reliable array of partitioned services
- RACS: A reliable array of cluster-structured server processes



RACS: Two perspectives

- A load-balancer (might be hardware) in front of a set of replicas, but with "affinity" mechanism
- A partitioning function (probably software), then random choice within replicas



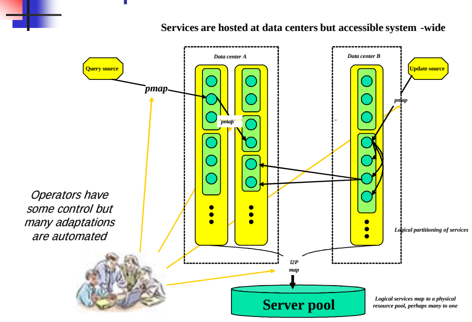
Affinity

- Problem is that many clients will talk to a service over a period of time
  - Think: Amazon.com, series of clicks to pick the digital camera you prefer
- This builds a "history" associated with recent interactions, and cached data
- We say that any server with the history has an *affinity* for subsequent requests

Affinity issues favor *pmap*

- Hardware load balancers are very fast
  - But can be hard to customize
  - Affinity will often be "keyed" by some form of content in request
  - HLB would need to hunt inside the request, find the content, then do mapping
  - Easy to implement in software… and machines are getting very fast…

Our platform in a datacenter



Problems we'll now face

- The single client wants to talk to the "correct" server, but discovers the service by a single name.
  - How can we implement pmap?
- We need to replicate data within a partition
  - How should we solve this problem?
- Web Services don't tackle this

More problems

- Our system is complex
  - How to administer?
  - How should the system sense load changes
  - Can we vary the sizes of partitions?
  - How much can be automated?
  - To what degree can we standardize the architecture?
  - What if something fails?

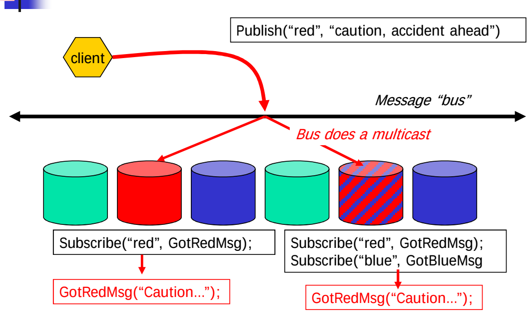
Event "notification" in WS

- Both CORBA and Web Services tackle just a small subset of these issues
- They do so through a
  - Notification (publish-subscribe) option
  - Notification comes in two flavors; we'll focus on just one of them (WS_NOTIFICATION)
  - Can be combined with "reliable" event queuing
- Very visible to you as the developer:
  - Notification and reliable queuing require "optional" software (must buy it) and work by the developer.
  - Not trivial to combine the two mechanisms

Publish-subscribe basics

- Dates to late 1980's, work at Stanford, Cornell, then commercialized by TIBCO and ISIS
- Support an interface like this:
  - Publish("topic", "message")
  - Subscribe("topic", handler)
- On match, platform calls handler(msg)

Publish-subscribe basics



WS_NOTIFICATION

- In Web Services, this is one of two standards for describing a message bus
  - The other is a combination of WS_EVENTING and WS_NAMING but seems to be getting less "traction"
- Also includes "content filtering" after receipt of message
- No reliability guarantees

How it works

- WS-Notification and WS-Eventing both assume that there is a server running the event notification system
  - To publish a message, send it to the server
  - To subscribe, tell the server what you are interested in
  - The server does the match-making and sends you matching messages

A brief aside (a complaint)

- Indirection through a server is slow
- Many pub-sub systems let data flow directly from publish to subscriber, for example using UDP multicast
- But WS-Notification and WS-Eventing don't allow that pattern. This seems to be an oversight by the standards group.

Content filtering

- Basic idea is simple
  - First deliver the message based on topic
  - But then apply an XML query to the message
  - Discard any message that doesn't match
- Application sees only messages that match both topic and query
- But costs of doing the query can be big

What about reliability?

- Publish-subscribe technologies are usually reliable, but the details vary
  - For example, TIB message bus will retry for 90 seconds, then discard a message if some receiver isn't acknowledging receipt
  - And some approaches assume that the receiver, not the sender, is responsible for reliability
- In big data centers, a source of trouble

Broadcast Storms

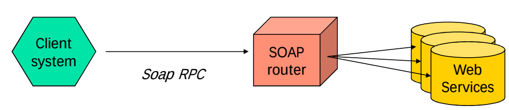
- A phenomenon of high loss rates seen when message bus is under heavy load
  - Requires very fast network hardware and multiple senders
  - With multicast, can get many back-to-back incoming messages at some receivers
  - These get overwhelmed and drop messages, must solicit retransmission
  - The retransmissions now swamp the bus
- Storms can cause network "blackouts" for extended periods (minutes)!

What about WS_RELIABILITY?

- Many people naïvely assume that this standard will eliminate problems of the sort just described
- Not so!
  - WS_RELIABILITY "looks" like it matches the issue
  - But in fact is concerned with a different problem....

Recall our naïve WS picture

- What happens if the Web Service isn't continuously available?
  - Router could reject request
  - But some argue for "message queuing"



Message queuing middleware

- A major product category
  - IBM MQSeries, HP MessageQueue, etc
  - Dates back to early client-server period when talking to mainframes was a challenge
  - Idea: Client does an RPC to “queue” request in a server, which then hands a batch of work to the mainframe, collects replies and queues them
  - Client later picks up reply

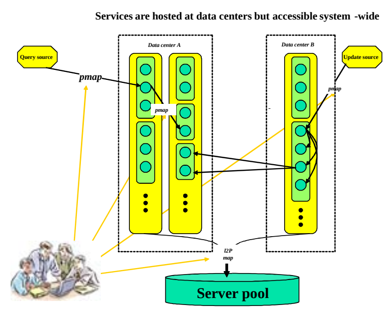
WS_RELIABILITY

- This standard is “about” message queuing middleware
  - It allows the client to specify behavior in the event that something fails and later restarts
    - At most once: easiest to implement
    - At least once: requires disk logging
    - Exactly once: requires complex protocol and special server features. Not always available

Can a message bus be reliable?

- Publish-subscribe systems don’t normally support this reliability model
- Putting a message queue “in front” of a message bus won’t help
  - Unclear who, if anyone, is “supposed” to receive a message when using pub-sub
  - The bus bases reliability on current subscribers, not “desired behavior”

Back to our data center



Back to our data center

- We’re finding many gaps between what Web Services offer and what we need!
- Good news?
  - Many of the mechanisms do exist
- Bad news?
  - They don’t seem to fit together to solve our problem!
  - Developers would need to hack around this

Where do we go from here?

- We need to dive down to basics
- Understand:
  - What does it take to build a trustworthy distributed computing system?
  - How do the technologies really work?
  - Can we retrofit solutions into Web Services?
- Our goal? A “scalable, trustworthy, services development framework”.