CS514: Intermediate Course in Operating Systems

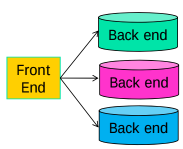
Professor Ken Birman
Vivek Vishnumurthy: TA

Real-world time-critical systems

- The challenge:
  - Suppose I need to build a rapidly responsive system
  - I want to handle large scale
  - I plan to use a modular architecture
- Can this be done in a web services setting?

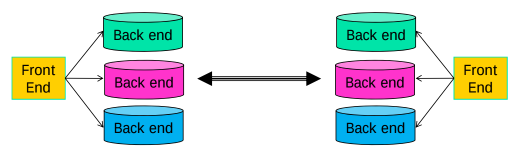
A "system of systems"

- We use the term "system of systems" or SoS to capture this concept
- Examples will help clarify the idea
- Basic structure:



A "system of systems"

- Or might interconnect systems at different data centers to give a reasonably integrated "picture"



Examples: Amazon

- Amazon would often use the front end to build a web page for a user
- The back-end systems fill in content
  - Product popularity
  - Current inventory
  - Great deals on related products
  - Products other people who did a similar search ultimately purchased…

Why is this "time critical"?

- Amazon is graded by quick accurate response
  - Good grade: You buy the book
  - Bad grade: You use Google and shop elsewhere
- For Amazon's line of business, this SoS configuration is as critical as it gets!

Akamai

- Corporate site controls a large number of satellite systems
- Goal: Move content to be close to users who are likely to access that content
- Time critical aspect: Akamai is paid by hosts seeking to ensure snappy load times for their web sites

Military example

- Team comes under fire, calls for help
- Commander needs to know
  - What resources are available?
  - What's the terrain
  - Where have enemy forces been seen?
  - Is there an evacuation option?
- ... and needs a fast response

Air Traffic Control Example

- New radar ping detected
  - Track formation system should fit this to existing tracks (or create a new one)
  - Flight plan lookup should check for known aircraft that might match this track
  - Warnings system should check for proximity rules
  - Long term planner should schedule a landing slot

Air Traffic Control Example

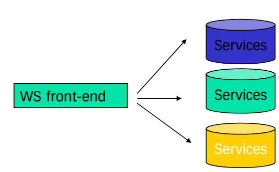
- Also see issues from controller to controller
  - When A hands off to B need to ensure continuous coverage
- And when centers talk to each other
  - France has 5 ATC centers... Europe has hundreds...

Issues? Let's focus on scaling

- Scalability allows us to handle more load and also provides fault-tolerance
  - Each service becomes a replicated group of servers that cooperate
  - They replicate data by multicasting updates
  - And the reads are load-balanced
- Issues are specific to time-criticality?

Tempest

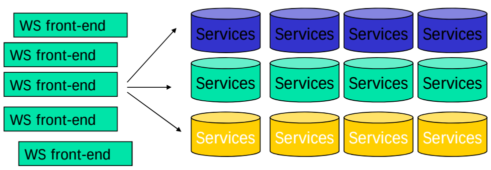
- Start with a standard web services application
- Perhaps, builds web pages for air traffic controller



Castor 4/07

Tempest

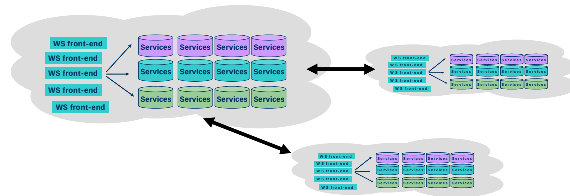
- We'll scale it out by replicating the components… and automate management, repair, adaptation even when faults occur



Castor 4/07

Tempest

- Then interconnect data centers



Castor 4/07

How to solve such problems?

- Tools in our toolkit
  - UDP multicast – very fast, unreliable
  - RON – routes around problems, unreliable
  - BitTorrent – receivers cooperate to offload work from the sender
  - Virtual synchrony – strong consistency
  - Quorums – even stronger (but slower)
  - CASD or Ricochet: real-time multicast

Too many choices!

- Need to ask
  - How strong does the consistency property need to be for the application of interest?
  - How harsh is the runtime environment?
  - How critical is timing?
  - Is the system "safe" if the primitive is unreliable?

How would Amazon answer?

- To guarantee fast response, they bought lots of hardware
  - … now they damn well expect speedups!
- Selling a book that is actually out of stock isn't a disaster
- Fast matters more than "real time" of the provable, conservative kind

Best technology for Amazon?

- Probably something like Ricochet would work best for them
  - Gets the update through FAST
  - Uses pro-active FEC to recover from likely patterns of loss
  - Background gossip mechanism repairs any losses not caught by FEC
- How might inconsistency "look" to users?

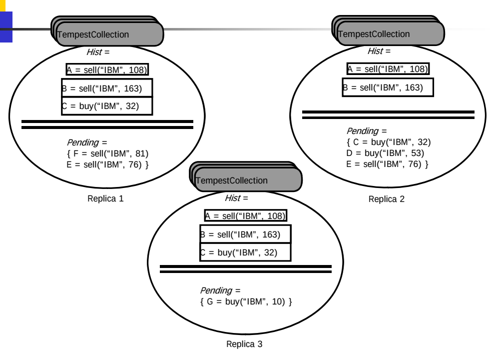
Consistency in Tempest

- Recall that transactional services offer strong data consistency model
  - each read operation returns the result of the latest write
- Tempest implements a weaker model called *sequential consistency*
  - every replica sees the operations on the same data item in the same order
  - order may be different than the order updates were issued

Tempest Collections

- Persistent service state = collection of objects
- Each object (*obj*) is naturally represented by the tuple $\langle Hist_{obj}, Pending_{obj} \rangle$
  - *Hist* is the state of the object
    - current value or list of updates
  - *Pending* is the set of updates that cannot be applied yet
    - applied when ordering consistent across

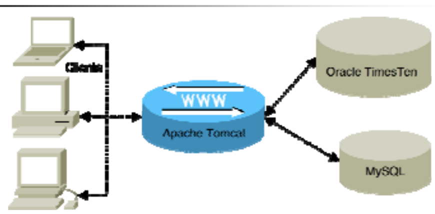
A Tempest Service



Two level implementation

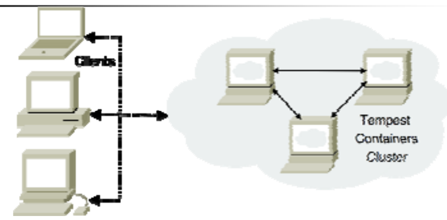
- To do a read, load-balance on some randomly picked component
  - Access the persistent state of the collection
- To do a write
  - Multicast the update with Ricochet.
  - On arrival, update goes on "pending queue"
  - Periodically, multicast an ordering to use
  - Run a background gossip protocol to clean up any lingering inconsistencies

Evaluation



Baseline: Same service using in memory database, and a transactional ACID database engine

Evaluation

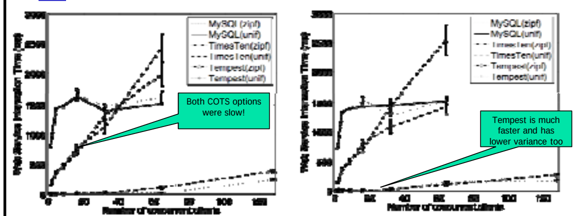


Tempest configuration: clients multicast requests to a group of processes using Ricochet

Experiment

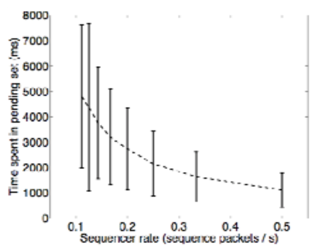
- clients issue requests at various rates
- request distributions read / write intensive
- startup phase, populate with 1024 objects
- request distribution uniform or zipf
- each client performs 10 requests/s
- results averaged over 10000 runs/client

Performance



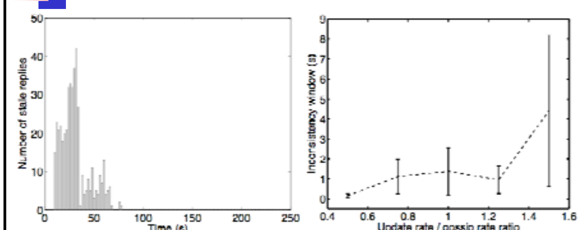
Request latency - on left write intensive, on right read intensive

Delay to order pending updates



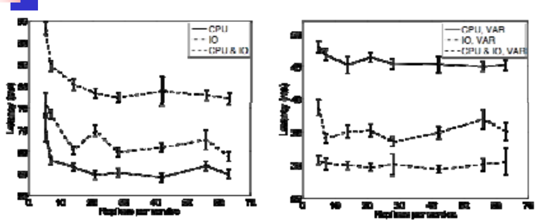
Pending set residency time, update rate 1/200 ms

Recovery under load



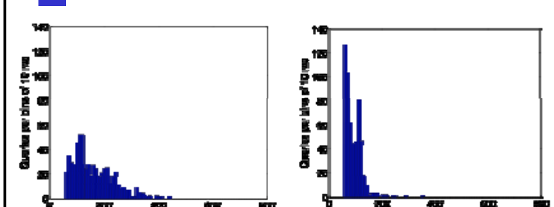
Behavior of affected replicas during a 40 second disruption

Services characteristics



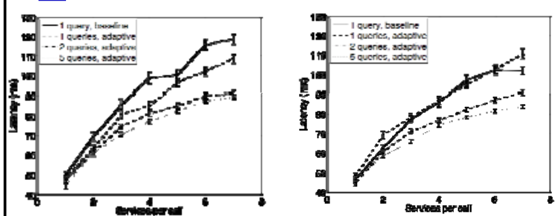
Individual service response time. Left - services with small response time variance, right large.

PetStore



Response time histogram. Left: services not replicated, right each service replicated 8 times.

PetStore



Left: 5 replicas, right: 8 replicas

Summary

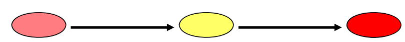
- Tempest framework can support time critical services
  - Model matches what “Amazon wants”
- Developers need not worry about scalability, fault-tolerance
- Tempest automatically adapts & reacts to load fluctuations and failures
- Adding inter-datacenter features now

What would an Air Traffic System want?

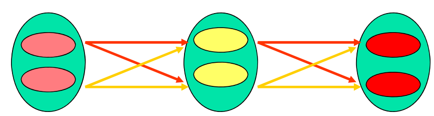
- Here, we need stronger consistency for many purposes
  - For example, system will hide any failure without loss of timing properties
  - And timing properties will be “extremely good”

Replicated components

- Pipelined computation



- Transformed computation



Choice we saw last time

- Could use CASD
  - Benefit: provable timing properties, ordering, reliability
  - Weakness: For high quality, very slow
- Could also consider Virtual Synchrony
  - Benefit: Strong consistency
  - Weakness: Fast, but can’t guarantee timing properties

More choices

- What about using consensus (Paxos)?
  - Here we would get very strong lock-step guarantees
  - Even if a node fails, state it saw is guaranteed to be correct
  - But even slower

How would we pick?

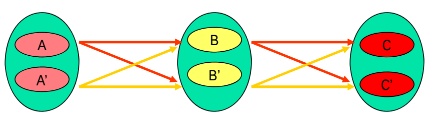
- Need to ask how application "balances" requirements
- Actual situation for an ATC system?
  - Consistency is extremely important
  - Also want speed, but not necessarily real-time of a provable kind
  - Hence would look at Paxos versus Virtual SYnchrony

Picking between Paxos and Vsync

- Virtual synchrony isn't safe enough!
  - Issue is that if a controller is told "ok to route plane X into sector Y", we'll take an action that can't be undone
- Hence Paxos guarantee is required
  - Either use the actual Paxos algorithm
  - Or use virtual synchrony in the "safe" (flushed) mode
  - Yes, this is slower... but it is also safer!

More practical questions

- How does one deal with requests in chains of replicated components?



- When A talks to B, B and B' will each see duplicated request from A and A'

Challenges of request duplication

- Must be careful to ensure that A and A' are deterministic!
  - Threads, timers, reading the clock, looking at the environment, even reading I/O from multiple sources can all make a program non-deterministic
  - In this case A and A' could deviate!
- Forces an unnatural coding style

Then....

- Suffices to number operations
- B and B' expect duplicates but don't wait for them
  - Take first incoming request
  - Discard duplicate (if we get one)

Raises a question

- Suppose we are doing read-only requests
- Is it best to send a request ONCE?
  - We can spread the load evenly
  - But sometimes may hit a busy node and get a long delay
- ...Or more than once?
  - Loads the service more... but maybe reply comes back sooner!

Generalized question

- For a system like Tempest
  - How much should each service be replicated to ensure best timing properties?
    - Tradeoff: Overhead versus benefit from light loads on queries
    - Answer may vary from service to service
  - How best to handle real-only requests
  - How to handle a transient like a load surge or a node failing

Summary

- Many real-world systems need time-critical functionality
- In systems of systems, this is tricky!
  - Forces tradeoffs: speed, versus consistency
  - Stronger properties are usually slower… but are genuinely safer!
- Smart designers are forced to really think the issues through step-by-step!