CS514: Intermediate Course in Operating Systems

Professor Ken Birman
Vivek Vishnumurthy: TA

Applications of these ideas

- Over the past three weeks we've heard about
  - Transactions
  - Logical time, consistent cuts
  - Distributed checkpoint mechanisms
  - Agreement protocols, such as 2PC and 3PC
- Before we drill down and learn more mechanisms like these… what good are these things?

Applications of these ideas

- Today, we'll review some practical applications of the mechanisms we've studied
  - Each is representative of a class
  - Goal is to illustrate the wide scope of these mechanisms, their power, and the ways you might use them in your own work

Specific topics we'll cover

- Transactions on multiple servers
- Auditing a system
- Detecting deadlocks
- Fault-tolerance in a long-running eScience application
- Distributed garbage collection
- Rebalancing objects in a cluster
- Computing a good routing structure
- Migrating tasks and TCP connections from server to server

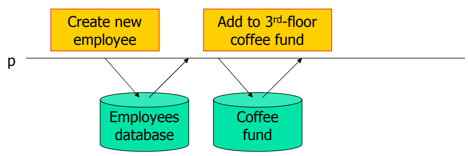
Solutions "inspired" by paradigms

- The solutions we arrive at won't always use the earlier protocol in a literal way
- Think of "paradigms" or "templates"
  - When they match a problem…
  - … you can specialize them to create a good solution

Let's start with 2PC and transactions

- The problem:
  - Some applications perform operations on multiple databases
  - We would like a guarantee that either *all* the databases get updated, or *none* does
- The relevant paradigm? 2PC

Problem: Pictorial version



- Goal? Either p succeeds, and both lists get updated, or something fails and neither does

Issues?

- P could crash part way through…
- … a database could throw an exception, e.g. "invalid SSN" or "duplicate record"
- … a database could crash, then restart, and may have "forgotten" uncommitted updates (presumed abort)

2PC is a good match!

- Adopt the view that each database votes on its willingness to commit
  - Until the commit actually occurs, update is considered temporary
  - In fact, database is permitted to discard a pending update (covers crash/restart case)
- 2PC covers all of these cases

Solution

- P runs the transactions, but warns databases that these are part of a transaction on multiple databases
  - They need to retain locks & logs
- When finished, run a 2PC protocol
  - Until they vote "ok" a database can abort
- 2PC decides outcome and informs them

Low availability?

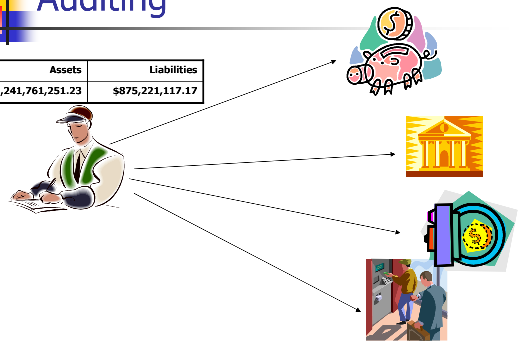
- One concern: we know that 2PC blocks
  - The failure scenario isn't all that rare
- Options?
  - Could use 3PC to reduce this risk, but will pay a cost on every transaction
  - Or just accept the risk
  - Can eliminate the risk with special hardware but may pay a fortune!

Next example: Auditing

- Our goal is to "audit" a system
  - Involves producing a summary of the state
  - Should look as if system was idle
- Our options?
  - Freeze the whole system, then snapshot the state. This is very disruptive
  - Or could use the Chandy/Lamport "consistent snapshot" algorithm

Auditing

Assets	Liabilities
$1,241,761,251.23	$875,221,117.17



Uses for auditing

- In a bank, may be the only practical way to understand "institutional risk"
  - Need to capture state at some instant in time. If branches report status at closing time, a bank that operates world-wide gets inconsistent answers!
- In a security-conscious system, might audit to try and identify source of a leak
- In a hospital, want ability to find out which people examined which records
- In an airline, might want to know about maintenance needs, spare parts inventory

Practical worries

- Snapshot won't occur at a point in *real* time
  - Could be noticeable to certain kinds of auditors
  - In some situations only a truly instantaneous audit can be accepted, but this isn't common
- What belongs in the snapshot?
  - Local states… but handling messages in transit can be very hard to do
  - With Web Services, some forms of remote server calls are nearly transparent!

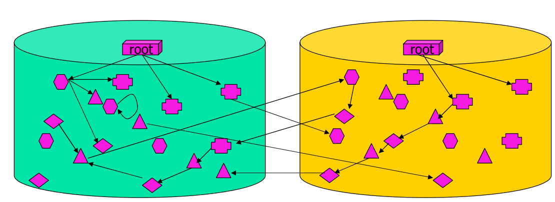
Pros and cons

- Implementing the algorithm is only practical
    - If we can implement lossless FIFO channels…
    - … and can intercept and log messages
    - … and in this way capture a "fake" instant in time,
  - So if all of that works… we're golden
- But doing so may be challenging
  - WS doesn't really have channels. So how can we interpose a logging mechanism between processes?
  - We'll be forced to implement a "user-level" version of the algorithm by structuring the application itself appropriately
  - This is one of our project assignments

Next in our list?

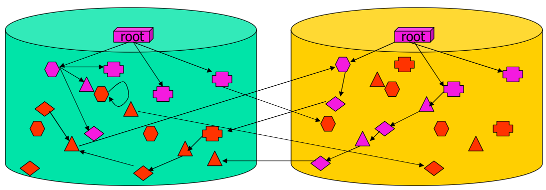
- Distributed garbage collection
  - Arises more and more commonly in big systems!
- The problem: applications create objects and pass one-another URLs
  - We would like to delete objects if there are no URLs pointing to them in the system
  - We'll assume that these URLs all live in machine-readable objects in the file system
  - Assume we know "root" object in each file system (and are given a list of file systems)

Distributed Garbage Collection



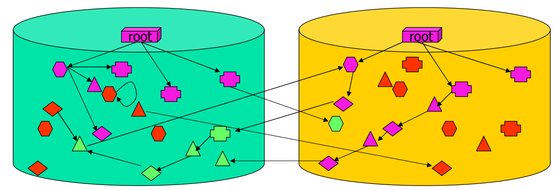
Distributed Garbage Collection

- Make a list of candidates at each site



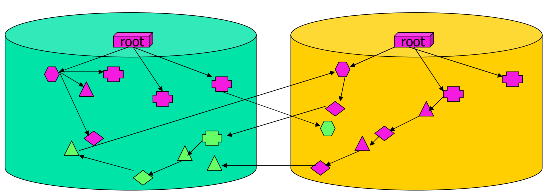
Distributed Garbage Collection

- Remove items from that list if there is a live pointer to them from a remote site



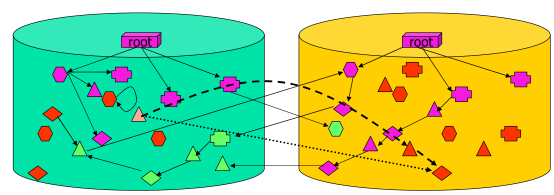
Distributed Garbage Collection

- Delete dead items



Distributed Garbage Collection

- Object being moved can confuse our search. So can a crashed file server



Review: what makes it hard?

- Must detect cycles and delete them if disconnected from a root (any root)
- While the algorithm is running, objects might be getting created or deleted or moved
- Machines may be down, preventing us from visiting them. And when a machine recovers after being down for a long time, it may "start" by moving objects to other machines

Conceptual solution

- A consistent snapshot might do the trick
  - The "local snapshot" would be a list of objects that have no local references (candidates to garbage collect), and URLs pointing to remote objects
    - No need to "announce" existence of an object that has at least one local reference
  - Channel state needs to show objects being moved from machine to machine
  - Then build a graph, mark nodes reachable from any root. Delete unreachable nodes

Challenges

- Dealing with failed machines is hard
  - Our algorithm needs to hang around waiting for such a machine to restart
  - But normally, in Web Services, there is no system-wide "machine status" service and messages sent to a machine while it is down will time out and be discarded!
  - Emblematic of lack of a "life cycle" service for applications in Web Services systems

Options?

- Best option might be to create your own object management subsystem
  - Applications would need to use it when creating, deleting, or moving objects
  - It could then implement desired functionality at the user level
- Another possibility: garbage collector could just retry until it manages to run when all machines are healthy

More practical issues

- In big centers some machines get added and some are retired, and this can be abrupt
  - So how do we deal with references both to and from such a machine?
  - Seems like a "group join" problem
- In very large systems, centralized snapshot scheme might not scale

Other consistent snapshot uses?

- Detecting distributed deadlocks
  - Here we build a "waiting for" graph
- Rebalancing objects in a cluster
  - Snapshot used to capture state
  - Then some process figures out an ideal layout and, if necessary, initiates object movement
- Computing a good routing structure
  - If we base on a snapshot can guarantee consistency (provided that everyone switches routing tables at the same time, of course)

Review of practical challenges

- We've seen that it can be very hard to interpose our algorithm in a complex system
  - Particularly if that system has pre-existing components that we can't access
  - For example many products have services of their own inside and we can't modify them
  - But can often hack-around this by building application-level libraries ("layers") to "wrap" the handling of the objects we're interested in

Review of practical challenges

- Who should run the algorithm?
  - Our algorithm is described as if there was a leader "in charge" of running it
    - The leader starts the protocol execution
    - Then collects the results up
    - Computes the desired "actions"
    - And tells everyone what to do
  - But where did the leader come from?

Leaders in snapshot protocols

- Many systems just punt on this issue
  - For example, might say that "any process can initiate a snapshot"
  - But now must deal with having two running at the same time
- Why is this hard?
  - Must somehow "name" each protocol instance
  - Keep results separated by instance id
  - For example: (leader's IP address, unique #)

Fault-tolerance concerns

- We mentioned difficulties created by failed machines
  - First, in Web Services, there is no standard way to recognize that a machine is faulty
    - Timeouts can give inconsistent results
  - Second, we lack any way to make sure a recovering machine will do something "first"
    - If we had such an option we might be able to use it to hack around our needs
  - In practice many protocols run to completion but "fail" if some machines didn't respond in time

Fault-tolerance concerns

- What if the leader who started the protocol fails while it's running?
  - Logs will start to get very big. And system runs slowly while logging
  - One option: throw away snapshot data after some amount of time elapses
    - But how long to wait?
    - Any fixed number could be a bad choice…

Concurrently running instances

- If we uniquely identify each protocol instance we can (potentially) run multiple instances in parallel
  - Requires clever data structures and bookkeeping
  - For example, in systems where messages might be large, might not want any single message to end up in multiple logs

In your banking project…

- We'll ask you to tackle this set of issues
- For your purposes, state is small and this helps quite a bit
- The best solutions will be
  - Tolerant of failures of branches
  - Tolerant of failures in the initiator processes
  - Able to run multiple concurrent snapshots

Last topic for today

- We have two additional problems on our list
  - Fault-tolerance in a long-running eScience application
  - Migrating work in an active system
- Both are instances of checkpoint-restart paradigm

eScience systems

- Emergence of the global grid has shifted scientific computing emphasis
  - In the past focus was on massive parallel machines in supercomputing centers
  - With this new shift, focus is on clustered computing platforms
  - SETI@Home illustrates grid concept: here the cluster is a huge number of home machines linked by the Internet!

The fault-tolerance problem

- Some applications are embarassingly parallel
  - The task decomposes into a huge number of separate subtasks that can be farmed out to separate machines, and they don't talk to each other
  - Here, we cope with faults by just asking someone else to compute any result that doesn't show up in reasonable time

The fault-tolerance problem

- But not all applications fit that model
  - IBM Blue Gene computer has millions of CPUs
  - Applications often communicate heavily with one-another
  - In this environment, we can't use the "zillions of separate tasks" model
- Researchers are doing coordinated checkpointing

How they approach it

- Recent trends favor automatically doing checkpoint/restart with the compiler!
  - The compiler can analyze program state and figure out how to make a checkpoint and how to incrementally update it
  - The runtime environment can log channel state and handle node-to-node coordination
- Effect is to hide the fault-tolerance mechanism from the programmer!

This is a big project at Cornell

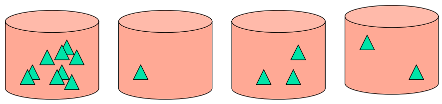
- Keshav Pingali and Paul Stoghill lead it
  - They are working in the Web Services paradigm, but compiling codes that were written using MPI and PVM
  - Solutions have extremely low overhead
    - Often below 1% slowdown
  - And are very transparent
    - Potentially completely invisible to the user

Task *migration*

- A closely related problem
- Arises in systems where some actions turn out to be very long running and can't be "detected" ahead of time
  - For example, computing an audit
  - Reorganizing a database
  - Defragmenting a disk…

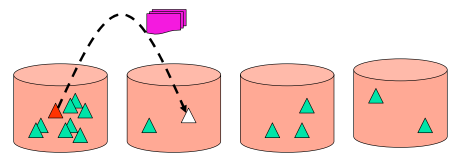
Scenario

- We accept tasks over the network and load-balance them onto servers
- But some servers get overloaded



Scenario

- Idea is to checkpoint the individual task
- Then move to some other place and restart from the checkpoint



What makes this hard?

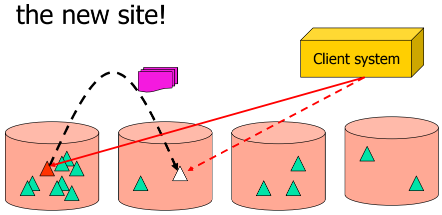
- We've shown our single task as if it was on one machine
  - But in modern systems one "task" may have activities on multiple computers
    - For example if a program talks to remote servers and they maintain some state for it
    - A checkpoint of such a task is a bit like a consistent snapshot!

What makes it hard?

- We've also drawn the picture as if each task is disconnected from the rest of the world
  - But many tasks will actually have "clients" that interact with the task over time
  - If we move the task… we need to move the associated "communication endpoint"
  - And the clients will need to know

Scenario

- A task with a client talking to it. The client's "endpoint" will need to shift to the new site!



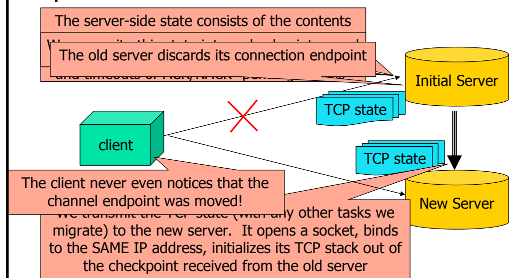
Why is this hard to do?

- In Web Services (and CORBA) that client is probably connected to the site via TCP
  - Must either have a way to ask the client to reconnect, which may be impractical
  - Or a way to transparently move a TCP endpoint

Could one "move" a TCP connection?

- We would need to move the IP address
  - We know that in the modern internet, IP addresses do move, all the time
  - NATs and firewalls do this, why can't we?
- We would also need to move the TCP connection "state"
  - Depending on how TCP was implemented this may actually be easy!

Migrating a TCP connection



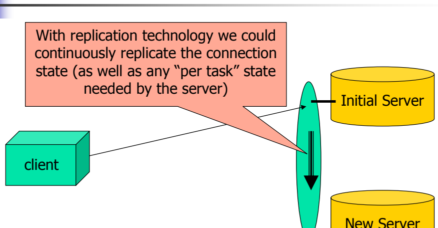
TCP connection state

- Includes:
  - The IP address, port # used by the client and the IP address and port on the server
    - Best to think of the server as temporarily exhibiting a "virtual address"
    - That address can be moved
  - Contents of the TCP "window"
    - We can write this down and move it too
  - ACK/NACK state, timeouts

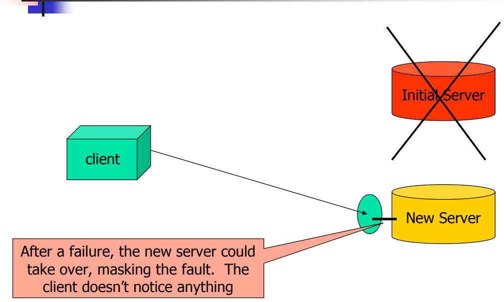
Generalizing the idea

- If we can checkpoint and move a TCP endpoint without breaking the channel
  - Then we can also imagine a continuously replicated TCP endpoint
  - This would let us have an unbreakable connection from one client to a cluster of servers!
  - We'll develop this idea later after we build reliable multicast primitives
- Value? Think of a "stock market ticker" that works, continuously, even if server crashes

Fault-tolerant TCP connection



Fault-tolerant TCP connection



Recap and summary

- We've begun to develop powerful, general tools
  - They aren't always of a form that the platform can (or should) standardize
  - But we can understand them as templates that can be specialized to our needs
  - Thinking this way lets us see that many practical questions are just instances of the templates we've touched on in the course

What next?

- We'll resume the development of primitives for replicating data
  - First, notion of group membership
  - Then fault-tolerant multicast
  - Then ordered multicast delivery
  - Finally leads to virtual synchrony "model"
  - Later will compare it to the State Machine replication model (quorums)
- And will use replication to tackle additional practical problems, much as we did today