Distributed Operating Systems, Con't

CS 239, Spring 1996

May 6, 1996

Outline of Introductory Materials

- Why distributed operating systems?
- Important issues in distributed OS's
- Important distributed OS tools and mechanisms

Important Issues in Distributed Operating Systems

- Communications model
- Process interaction
- Transparency
- Heterogeneity
- Autonomy
- Consistency and transactions

Autonomy

To some degree, users need to control their own resources

The more a system encourages interdependence, the less autonomy

How to best trade off sharing and interdependence versus autonomy?

Problems With Too Much Interdependence

- Vulnerability to failures
- Global control
- Hard to pinpoint responsibility
- Hard security problems

Problems With Too Much Autonomy

- Redundancy of functions
- Heterogeneity
  - Especially in software
- Poor resource sharing

Methods To Improve Autonomy

Without causing problems with sharing

- Replicate vital services on each machine
- Don't export services that are unnecessary
- Provide strong security guarantees

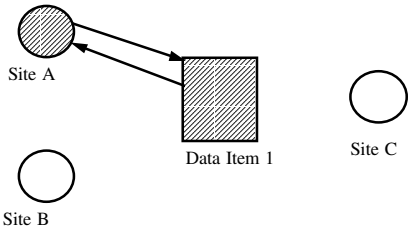
Consistency

Maintaining consistency is a major problem in distributed systems

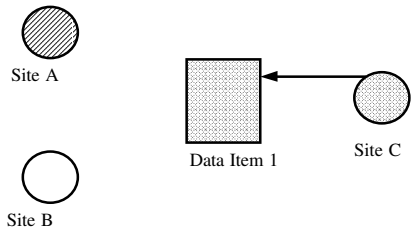
If more than one system accesses data, can be hard to ensure consistency

But if cooperating processes see inconsistent data, disasters are possible

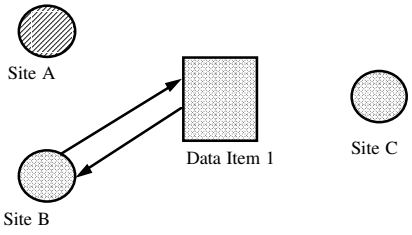
A Sample Consistency Problem



A Sample Consistency Problem, Con't



A Sample Consistency Problem, Concluded



Causes of Consistency Problems

- Failures and partitionings
- Caching effects
- Replication of data

So Why Do This Stuff?

Note these problems arise because of what are otherwise desirable features -

- Failures and partitioning
  - ➜Working in the face of failures
- Caching
  - ➜Avoiding repetition of expensive operations
- Replication
  - ➜Higher availability

Handling Consistency Problems

- Don't share data
  - – Generally not feasible
- Callbacks
- Invalidations
- Ignore the problem
  - – Sometimes OK, but not always

Callback Methods

Check that your data view is consistent whenever there might be a problem

In most general case, on every access

More practically, every so often

Extremely expensive if remote check required

High overheads if there's usually no problem

Invalidation Methods

When situations change, inform those who know about the old situation

Requires extensive bookkeeping

Practical in some cases when changes infrequent

High overheads if there's usually no problem

Consistency and Atomicity

Atomic actions are "all or nothing"

Either the entire set of actions occur

Or none of them do

At all times, including while being performed

Apparently indivisible and instantaneous

Relatively easy to provide in single machine systems

Atomic Actions in Single Processors

Lock all associated resources (with semaphore or other synchronization mechanism)

Perform all actions without examining unlocked resources

Unlock all resources

Real trick is to provide atomicity even if process is switched in the middle

Why Are Distributed Atomic Actions Hard?

Lack of centralized control

What if multiple processes on multiple machines want to perform an atomic action?

How to you properly lock everything?

How do you properly unlock everything?

Failure conditions especially hard

Important Distributed OS Tools and Mechanisms

- Caching and replication
- Transactions and two-phase commit
- Hierarchical name spaces
- Optimistic methods

Caching and Replication

Remotely accessing data is the pits

- It always takes longer
- It's less predictable
- It clogs the network
- It annoys other nodes
- Other nodes annoy you
- It's less secure

But What Else Can You Do?

Data must be shared

And by off-machine processes

If the data isn't local, and you need it, you must get it

So make sure data you need is local

The problem is that everyone else also wants their data local

Making Data Local

- Store what you need locally
- Make copies
- Migrate necessary data in
- Cache data
- Replicate data

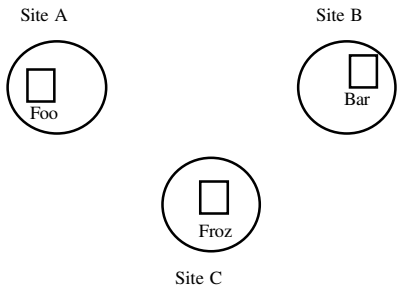
Store It Locally

Each site stores the data it needs on local media

But what if two sites need to store the same data?

Or if you don't have enough room for all your data?

Local Storage Example



Make Copies

Each site stores its own copy of the data it needs

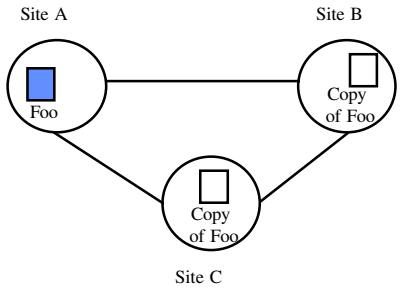
Works well for rarely updated data

- Like copies of system utility programs

Works poorly for frequently written data

Doesn't solve the problem of lack of local space

Copying Example



Migrate the Data In

When you need a piece of data, find it and bring it to your site

- Taking it away from the old site

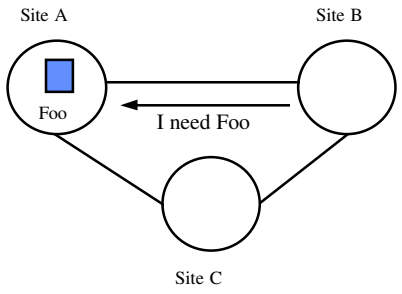
Works poorly for highly shared data

Can cause severe storage problems

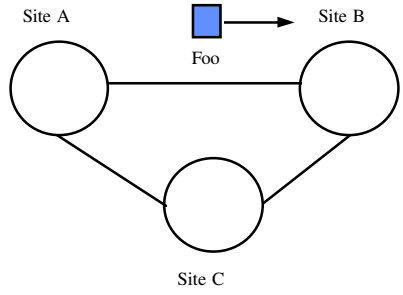
Can overburden the network

Essentially how shared software licenses work

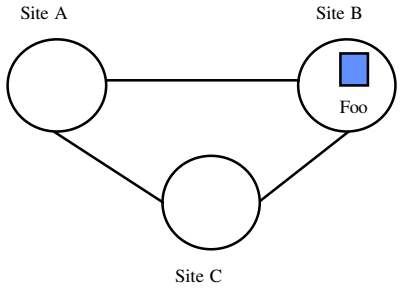
Migration Example



Migration Example, Continued



Migration Example, Concluded



Caching

When data is accessed remotely, temporarily store a copy of it locally

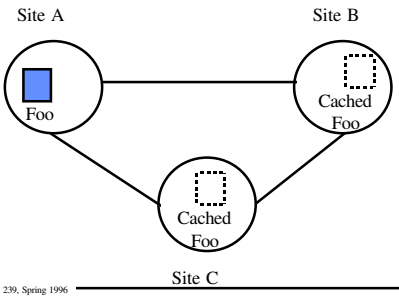
Perhaps using callback or invalidation for consistency

Or perhaps not

Avoids problems of storage

Still not quite right for frequently written data

Caching Example



Replication

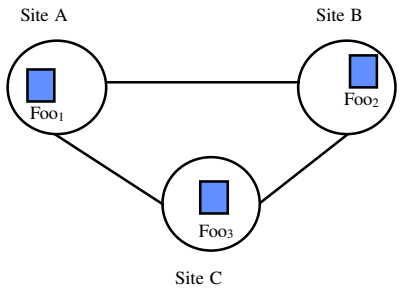
Maintain multiple logical replicas of the data

Changes made to one replica are automatically propagated to other replicas

Logically connects copies of data into a single entity

Doesn't answer question of limited space

Replication Example



Replication Advantages

- Most accesses to data are purely local
  So performance is good
- Fault tolerance
  Failure of a single node doesn't lose data
  Partitioned sites can access data
- Load balancing
  Replicas can share the work

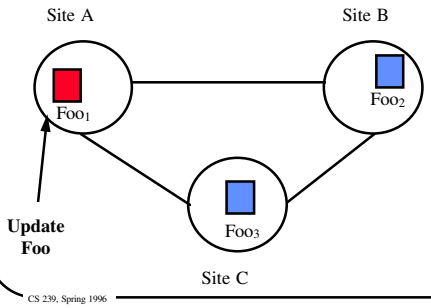
Replication and Updates

When a data item is replicated, updates to that item must be propagated to all replicas

Updates come to one replica

Something must assure they get to the others

Replication Update Example



Update Propagation Methods

- Instant versus delayed
- Synchronous versus asynchronous
- Atomic versus non-atomic

Instant Propagation Versus Delayed Propagation

"Instant" can't mean instant in a distributed system

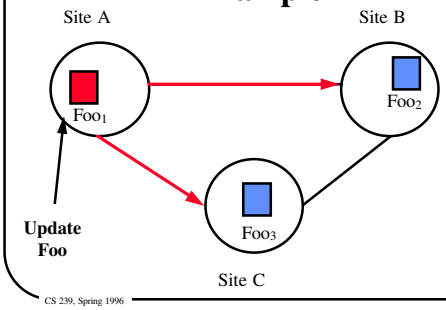
But it can mean "quickly"

Instant notification not always possible

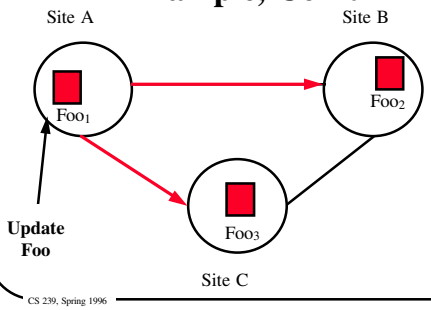
What if a site storing a replica is down?

So some delayed version of update is also required

Instant Update Propagation Example



Instant Update Propagation Example, Con't



Synchronous Versus Asynchronous Propagation

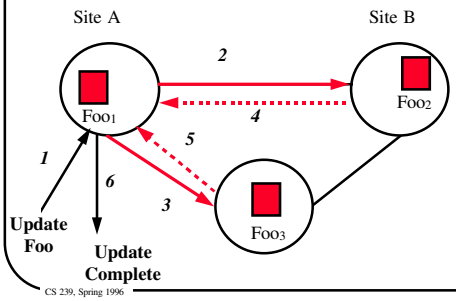
Update request sooner or later gets a success signal

Does it get it before all propagation completes (asynchronous) or not (synchronous)?

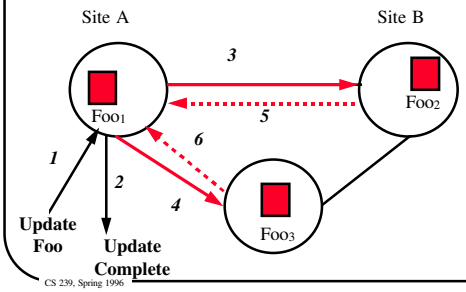
Synchronous propagation delays completion

Asynchronous propagation allows inconsistencies

Synchronous Propagation Example



Asynchronous Propagation Example



Atomic Versus Non-Atomic Update Propagation

Atomic propagation lets no one see new data until all replicas store it

Non-atomic lets users see data at some replicas before all replicas have updated it

Atomic update propagation can seriously delay data availability

Non-atomic propagation allows users to see potentially inconsistent data

Replication Consistency Problems

Unless update propagation is atomic, consistency problems can arise

One user sees a different data version than another user at the same time

But even atomic propagation isn't enough to prevent this situation

Concurrent Update

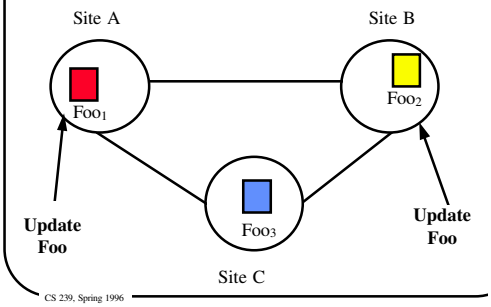
What if two users simultaneously ask to update different replicas of the data?

"Simultaneously" has a looser definition in distributed systems

How do you prevent both from updating it?

Update propagation style offers no help

Concurrent Update Example



Preventing Concurrent Update

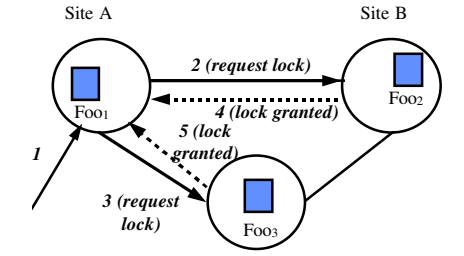
One solution is to lock all copies before making update

That's expensive

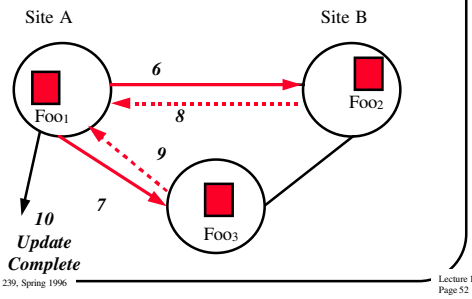
And what if one of 20 replicas is unavailable?

You must allow updates to data when partitions or failures occur

Locking Example



Locking Example, Continued



Problems With Locking

It's expensive

And what if one of 20 replicas is unavailable?

You must allow updates to data when partitions or failures occur

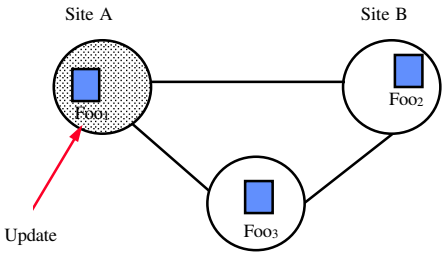
Concurrent Update Prevention Schemes

- Primary site
- Token approaches
- Majority voting
- Weighted voting

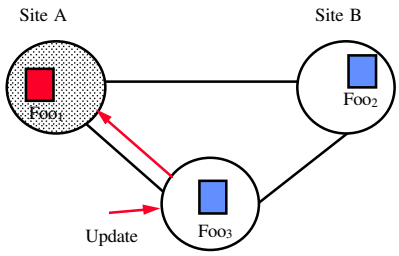
Primary Site Methods

Only one site can accept updates
  Or that site must approve all updates
In extraordinary circumstances, appoint new primary site
+ Simple
– Poor reliability, availability
– Non-democratic
– Poor performance in many cases

Primary Site Example



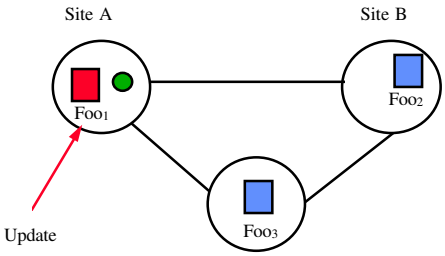
Second Primary Site Example



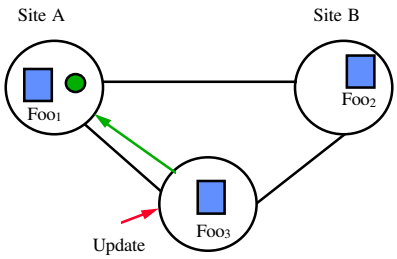
Token Based Approaches

Only the site holding the token can accept updates
But the token can move from site to site
+ Relatively simple
+ More adaptive than central site
– Poor reliability, availability
– Non-democratic
– Poor performance in some cases

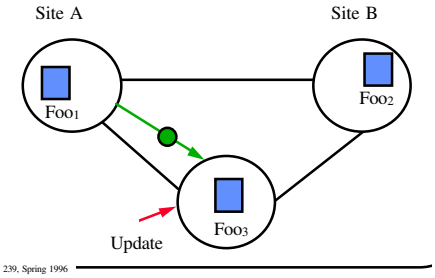
Token Example



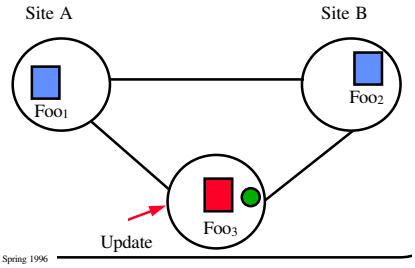
Second Token Example



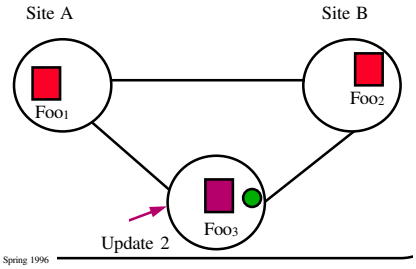
Second Token Example, Con't



Second Token Example, Concluded



Why is This Any Different Than Primary Site?



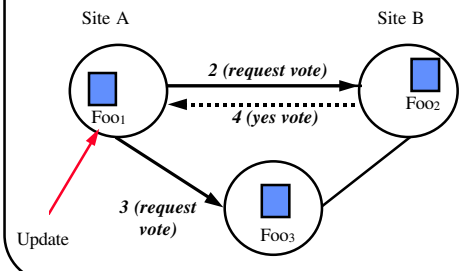
Majority Voting

To perform update, replica must receive approval from majority of all replicas

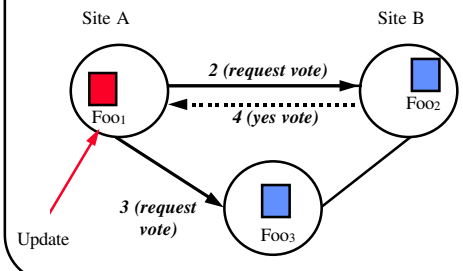
Once a replica grants approval to one update, it cannot grant it to another

Until the first update is completed

Majority Voting Example



Majority Voting Example



Majority Voting, Con't

+ Democratic
+ Easy to understand
+ More reliable, available
– Some sites still can't write
– Voting is a distributed action
  So it's expensive to do it

Weighted Voting

Like majority voting, but some replicas get more votes than others

Must obtain majority of votes, but not necessarily from majority of sites

Fits neatly into transaction models

Weighted Voting, Con't

+ More flexible than majority voting
+ Can provide better performance
– Somewhat less democratic
– Some sites still can't write
– Still potentially expensive
– More complex

Basic Problems With Update Control Methods

- Either very poor reliability/availability or expensive distributed algorithms for update
- Always some reliability/availability problems

Particularly bad for slow networks, expensive networks, flaky networks, mobile computers

Transactions

A special case of atomic actions
  Originally from databases

Sequence of operations that transform a current consistent state to a new consistent state
  Without ever exposing an inconsistent state

Transaction Example

Move $10 from my savings account to my checking account

Basically, subtract $10 from savings account, add $10 to checking account

But never "lose" my $10
  And never give me an extra $10

Running Transactions

Multiple transactions must not interfere

So you can run them one at a time

Or run them simultaneously

But avoiding all interference

Serializability avoids interference

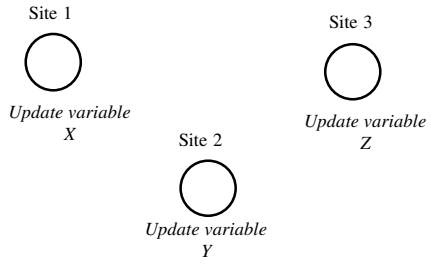
Serializability

A property of a proposed schedule of transactions

A serializable schedule produces the same results as some serial execution of those transactions

Even though the actions may have been performed in a different order

Consistency Example for a Distributed System



What Problems Could Arise?

- Other processes could write the variables
- Other processes could read the variables
- Failures could interrupt the process

How can we avoid these problems?