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 group communication
  - Process groups
  - Membership tracking and reporting "new views"
  - Reliable multicast, ordered in various ways
  - Dynamic uniformity (safety), quorum protocols
- So we know how to build group multicast… but 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

- Wrappers and Toolkits
- Distributed Programming Languages
- Wrapping a Simple RPC server
- Wrapping a Web Site
- Hardening Other Aspects of the Web
- Unbreakable Stream Connections
- Reliable Distributed Shared Memory

What should the user "see"?

- Presentation of group communication tools to end users has been a controversial topic for decades!
- Some schools of thought:
  - Direct interface for creating and using groups
  - Hide in a familiar abstraction like publish-subscribe or Windows event notification
  - Use inside something else, like a cluster mgt. platform a new programming language
- Each approach has pros and cons

Toolkits

- Most systems that offer group communication directly have toolkit interfaces
  - User sees a library with various calls and callbacks
  - These are organized into "tools"

Style of coding?

- User writes a program in Java, C, C++, C#...
- The program declares "handlers" for events like new views, arriving messages
- Then it joins groups and can send/receive multicasts
- Normally, it would also use threads to interact with users via a GUI or do other useful things

---

Toolkit ap

To send a multicast (here, a totally ordered one), you specify the group identifier from a join or lookup, a request code (an integer), and then the message. This multicast builds a message using a C-style format string. This abcast wants a reply from all members; the replies are floating poin

Here's got_msg. It gets invoked when a multicast arrived with the matching request code. This particular procedure extracts a string and an integer from the message and sends a reply. Abcast will collect all of those replies into a vector, set the caller's pointer to point to that vector, and return the number of replies it received (namely, the number of members in the current view)

- Join a group, state

```
Gid = pg_join(
      PG_INIT,
      XFER_IN,
```

- Multicast to a

```
nr = abcast(gi
```

- Register a call

```
isis_entry(REQ
```

- Receive a mult

```
void got_msg(message *mp) {
      Msg_scan("%s,%d", &astring, &anint);
      Reply(mp, "%f", 123.45);
}
```

Threading

- A tricky topic in Isis
  - The user needs threads, e.g. to deal with I/O from the client while also listening for incoming messages, or to accept new requests while waiting for replies to an RPC or multicast
  - But the user also needs to know that messages and new views are delivered in order, hence concurrent threads pose issues
- Solution? Isis acts like a "monitor" with threads, but running them one at a time unless the user explicitly "exits" the monitor

---

A tricky model to work with!

- We have…
  - Threads, which many people find tricky
  - Virtual synchrony, including choices of ordering
  - A new distributed "abstraction" (groups)
- Developers will be making lots of choices, some with big performance implications, and this is a negative

---

Examples of tools in toolkit

- Group join, state xfer
- Leader selection
- Holding a "token"
- Checkpointing a group
- Data replication
- Locking
- Primary-backup
- Load-balancing
- Distributed snapshot

How toolkits work

- They offer a programmer API
  - More procedures, e.g.
    - Create_replicated_data("name", type)
    - Lock_replica("name")
    - Update_replica("name", value)
    - V = (type)Read_replica("name")
  - Internally, these use groups & multicast
    - Perhaps, asynchronous cbcast as discussed last week…
    - Toolkit builder optimizes extensively, etc…

How programmers use toolkits

- Two main styles
  - Replicating a data structure
    - For example, "air traffic sector D-5"
    - Consists of all the data associated with that structure… could be quite elaborate
    - Processes sharing the structure could be very different (maybe not even the same language)
  - Replicating a service
    - For high availability, load-balancing

Experience is mixed….

- Note that many systems use group communication but don't offer "toolkits" to developers/end users
- Major toolkit successes include New York and Swiss Stock Exchange, French Air Traffic Control System, US AEGIS warship, various VLSI Fab systems, etc
  - But building them demanded special programmer expertise and knowledge of a large, complex platform
  - Not every tool works in every situation! Performance surprises & idiosyncratic behavior common. Toolkits never caught on the way that transactions became standard
- But there are several popular toolkits, like JGroups, Spread and Ensemble. Many people do use them

Leads to notion of "wrappers"

- Suppose that we could have a magic wand and wave it at some system component
  - "Replicatum transparentus!"
- Could we "automate" the use of tools and hide the details from programmers?

Wrapper examples

- Transparently…
  - Take an existing service and "wrap" it so as to replicate inputs, making it fault-tolerant
  - Take a file or database and "wrap" it so that it will be replicated for high availability
  - Take a communication channel and "wrap" it so that instead of connecting to a single server, it connects to a group

Experience with wrappers?

- Transparency isn't always a good thing
  - CORBA has a fault-tolerance wrapper
    - In CORBA, programs are "active objects"
    - The wrapper requires that these be deterministic objects with no GUI (e.g. servers)
    - CORBA replaces the object with a group, and uses abcast to send requests to the group.
      - Members do the same thing, "state machine" style
      - So replies are identical. Give the client the first one

Why CORBA f.tol. was a flop

- Users find the determinism assumption too constraining
  - Prevents use of threads, shared memory, system clock, timers, multiple I/O channels…
  - Real programs sometimes use these sorts of things *unknown to the programmer*
    - Who knows how the .NET I/O library was programmed by Microsoft? Could it have threads inside, or timers?
- Moreover, costs were high
  - Twice as much hardware… slower performance!
  - Also, had to purchase the technology separately from your basic ORB (and for almost same price)

Files and databases?

- Here, issue is that there are other ways to solve the same problem
  - A file, for example, could be put on a RAID file server
  - This provides high speed and high capacity and fault-tolerance too
  - Software replication can't easily compete

How about "TCP to a group?"

- This is a neat application and very interesting to discuss. We saw it in lecture 11. Let's look at it again, carefully

- Goals:
  - Client system runs standard, unchanged TCP
  - Server replaced by a group… leader owns the TCP endpoint but if it crashes, someone else takes over and client sees no disruption at all!

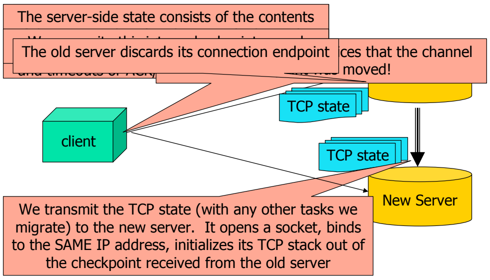
How would this work?

- Revisit idea from lecture 11
- Reminder: TCP is a kind of state machine
  - Events occur (incoming IP packets, timeouts, read/write requests from app)
  - These trigger "actions" (sending data packets, acks, nacks, retransmission)
  - We can potentially checkpoint the state of a TCP connection or even replicate it in realtime!

How to "move" a TCP connection

- We 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



The server-side state consists of the contents

The old server discards its connection endpoint

ces that the channel

moved!

We transmit the TCP state (with any other tasks we migrate) to the new server. It opens a socket, binds to the SAME IP address, initializes its TCP stack out of the checkpoint received from the old server

---

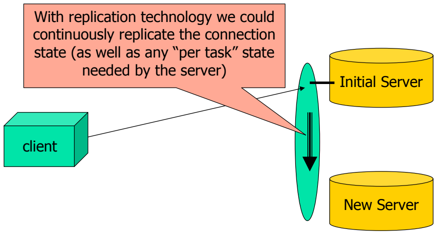
TCP connection state

- 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

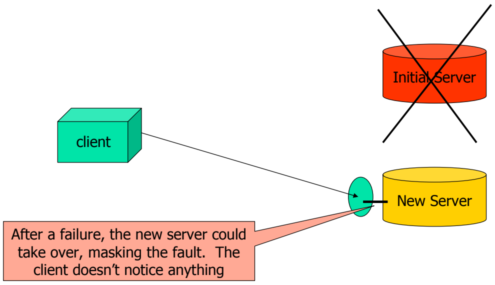
- Create a process group
  - Use multicasts when each event occurs (abcast)
  - All replicas can track state of the leader
  - Now if a new view shows that the leader has failed, a replica can take over by binding to the IP address

Fault-tolerant TCP connection



With replication technology we could continuously replicate the connection state (as well as any "per task" state needed by the server)

Fault-tolerant TCP connection



After a failure, the new server could take over, masking the fault. The client doesn't notice anything

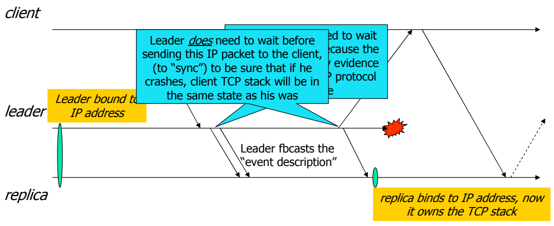
What's new?

- In lecture 11 we didn't know much about multicast… now we do
- This lets us ask how costly the solution would be
- In particular
  - Which multicast should be used?
  - When would a delay be incurred?

Choice of multicast

- We need to be sure that everyone sees events in the identical order
  - Sounds like abcast
- But in fact there is only a single sender at a time, namely the leader
  - Fbcast is actually adequate!
  - Advantage: leader doesn't need to multicast to itself, only to the replicas

Timeline picture



Leader *does* need to wait before sending this IP packet to the client, (to "sync") to be sure that if he crashes, client TCP stack will be in the same state as his was

Leader fbcasts the "event description"

replica binds to IP address, now it owns the TCP stack

Asynchronous multicast

- This term is used when we can send a multicast without waiting for replies
- Our example uses asynchronous fbcast
  - An especially cheap protocol: often just sends a UDP packet
  - Acks and so forth can happen later and be amortized over many multicasts
- “Sync” is slower: must wait for an ack
  - But often occurs in background while leader is processing the request, “hiding” the cost!

Sources of delay?

- Building event messages to represent TCP state, sending them
  - But this can occur concurrently with handing data to the application and letting it do whatever work is required
  - Unless TCP data is huge, delay is very small
- Synchronization before sending packets of any kind to client
  - Must be certain that replica is in the identical state

How visible will delay be?

- This version of TCP
  - May notice overhead for very small round-trip interactions: puts the sync event right in the measured RTT path
    - Although replica is probably close by with a very fast connection to the leader, whereas client is probably far away with a slow connection…
  - But could seem pretty much as fast as a normal TCP if the application runs for a long time, since that time will hide the delay of synchronizing leader with replica!

Using our solution?

- Now we can wrap a web site or some other service
  - Run one copy on each of two or more machines
  - Use our replicated TCP
- Application sees identical inputs and produces identical outputs…

Repeat of CORBA f.tol. idea?

- Not exactly…
  - We do need determinism with respect to the TCP inputs
  - But in fact we don’t need to legislate that “the application must be a deterministic object”
  - Users could, for example, use threads as long as they ensure that identical TCP inputs result in identical replies

Determinism worry

- Recall that CORBA transparently replicates objects
  - But insists that they be deterministic
  - And this was an unpopular requirement
- Our “Web Services wrapper” does too
  - But only requires determinism *with respect to the TCP inputs*
  - The server could be quite concurrent as long as its state and actions will be identical given same TCP request sequence: a less demanding requirement

Would users accept this?

- Unknown: This style of wrapping has never been explored in commercial products
- But the idea seems appealing… perhaps someone in the class will find out…

Distributed shared memory

- A new goal: software DSM
  - Looks like a memory-mapped file
  - But data is automatically replicated, so all users see identical content
- Requires a way for DSM server to intercept write operations

---

Some insights that might help

- Assume that programs have locality
  - In particular, that there tends to be one writer in a given DSM page at a time
  - Moreover, that both writers and readers get some form of locks first
- Why are these legitimate assumptions?
  - Lacking them, application would be highly non-deterministic and probably incorrect

---

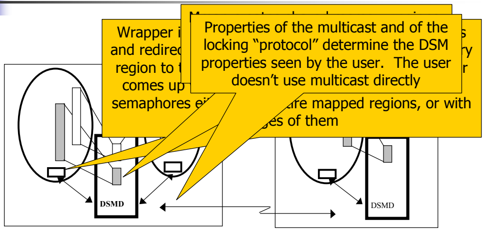
So what’s the model?

- Application “maps” a region of memory
- While running, it sometimes
  - Acquires a read or write lock
  - Then for a period of time reads or writes some part of the DSM (some “pages”)
  - Then releases the lock
- Gee… this is just our distributed replication model in a new form!

To implement this DSM…

- We need a way to
  - Implement the mapping
  - Detect that a page has become dirty
  - Invoke our communication primitives when a lock is requested or released
- Idea:
  - Use the Linux mapped file primitives and build a DSM “daemon” to send updates
  - Intercept Linux semaphore operations for synchronization

DSM with a daemon



Properties of the multicast and of the locking “protocol” determine the DSM properties seen by the user. The user doesn’t use multicast directly

Design choices?

- We need to decide how semaphores are associated with the mapped memory
  - E.g. could have one semaphore for the whole region; treat it as an exclusive lock
  - Or could have one per page
  - Could event implement a readers/writers mechanism, although this would depart from the Linux semaphore API

Design choices?

- Must also pick a *memory coherency model:*
  - Strong consistency: The DSM behaves like a single non-replicated memory
  - Weak consistency: The DSM can be highly inconsistent. Updates propagate after an unspecified and possibly long delay, and copies of the mapped region may differ
  - Release consistency (DASH project): Requires locking for mutual exclusion; consistent as long as locking is used
  - Causal consistency (Neiger and Hutto): If DSM update a → b, then b will observe the results of a.

Best choice?

- We should probably pick release consistency or causal consistency
  - Release consistency requires fbcast
  - Causal consistency would use cbcast
- The updates end up totally ordered along mutual exclusion paths and the primitive is strong enough to maintain this delivery ordering at all copies

False sharing

- One issue designer must worry about
  - Suppose multiple *independent* objects map to the same page but have distinct locks
  - In a traditional hardware DSM page ends up ping-ponging between the machines
  - In our solution, this just won't work!
- Our mechanism requires that there be one lock per "page"

Would this work?

- In fact it can work extremely well
  - In years past, students have implemented this form of DSM as a course project
  - Performance is remarkably good if the application "understands" the properties of the DSM
- Notice that DSM is really just a different API for offering multicast to user…

"Tools" we didn't discuss today

- Many people like publish-subscribe
  - Could just map topics to groups
  - But this requires that the group communication system scale extremely well in the numbers of groups, a property not all GCS platforms exhibit
  - Interesting current research topic
    - JGroups, Ensemble just have regular groups and can't handle apps that create millions of them
    - Spread tackles with "lightweight" groups… but his has some overheads (it delivers, then discards, extra msgs)
    - QuickSilver now investigating a new approach

Recap of today's lecture

- Wrappers and Toolkits
- Distributed Programming Languages
- Wrapping a Simple RPC server
- Wrapping a Web Site
- Hardening Other Aspects of the Web
- Unbreakable Stream Connections
- Reliable Distributed Shared Memory

… we've looked at each of these topics and seen that with a group multicast platform, the problem isn't hard to solve