CS514: Intermediate Course in Operating Systems

Professor Ken Birman
Vivek Vishnumurthy: TA

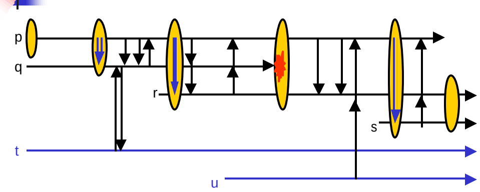
Virtual Synchrony

- A powerful programming model!
  - Called *virtual synchrony*
  - It offers
    - Process groups with state transfer, automated fault detection and membership reporting
    - Ordered reliable multicast, in several flavors
    - Extremely good performance

Why “virtual” synchrony?

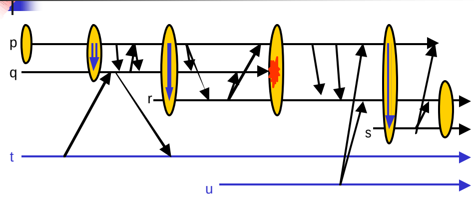
- What would a synchronous execution look like?
- In what ways is a “virtual” synchrony execution not the same thing?

A synchronous execution



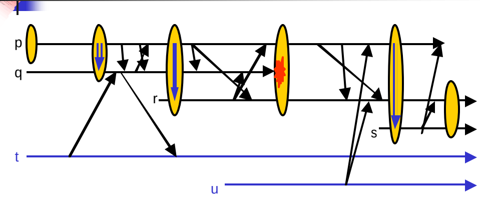
- With *true* synchrony executions run in genuine lock-step.

Virtual Synchrony at a glance



- With *virtual* synchrony executions only look “lock step” to the application

Virtual Synchrony at a glance



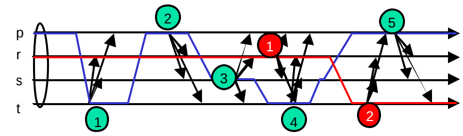
- We use the weakest (hence fastest) form of communication possible

Chances to "weaken" ordering

- Suppose that any conflicting updates are synchronized using some form of locking
  - Multicast sender will have mutual exclusion
  - Hence simply because we used locks, cbcast delivers conflicting updates in order they were performed!
- If our system ever *does* see concurrent multicasts… they must not have conflicted. So it won't matter if cbcast delivers them in different orders at different recipients!

Causally ordered updates

- Each thread corresponds to a different lock



- In effect: red "events" never conflict with green ones!

In general?

- Replace "safe" (dynamic uniformity) with a standard multicast when possible
- Replace abcast with cbcast
- Replace cbcast with fbcast
- Unless replies are needed, don't wait for replies to a multicast

Why "virtual" synchrony?

- The user sees what looks like a synchronous execution
  - Simplifies the developer's task
- But the actual execution is rather concurrent and asynchronous
  - Maximizes performance
  - Reduces risk that lock-step execution will trigger correlated failures

Correlated failures

- Why do we claim that virtual synchrony makes these less likely?
  - Recall that many programs are buggy
  - Often these are Heisenbugs (order sensitive)
- With lock-step execution each group member sees group events in identical order
  - So all die in unison
- With virtual synchrony orders differ
  - So an order-sensitive bug might only kill one group member!

Programming with groups

- Many systems just have one group
  - E.g. replicated bank servers
  - Cluster mimics one highly reliable server
- But we can also use groups at finer granularity
  - E.g. to replicate a shared data structure
  - Now one process might belong to many groups
- A further reason that different processes might see different inputs and event orders

Embedding groups into "tools"

- We can design a groups API:
  - pg_join(), pg_leave(), cbcast()...
- But we can also use groups to build other higher level mechanisms
  - Distributed algorithms, like snapshot
  - Fault-tolerant request execution
  - Publish-subscribe

Distributed algorithms

- Processes that might participate join an appropriate group
- Now the group view gives a simple leader election rule
  - Everyone sees the same members, in the same order, ranked by when they joined
  - Leader can be, e.g., the "oldest" process

Distributed algorithms

- A group can easily solve consensus
  - Leader multicasts: "what's your input"?
  - All reply: "Mine is 0. Mine is 1"
  - Initiator picks the most common value and multicasts that: the "decision value"
  - If the leader fails, the new leader just restarts the algorithm
- Puzzle: Does FLP apply here?

Distributed algorithms

- A group can easily do consistent snapshot algorithm
  - Either use cbcast throughout system, or build the algorithm over gbcast
  - Two phases:
    - Start snapshot: a first cbcast
    - Finished: a second cbcast, collect process states and channel logs

Distributed algorithms: Summary

- Leader election
- Consensus and other forms of agreement like voting
- Snapshots, hence deadlock detection, auditing, load balancing

More tools: fault-tolerance

- Suppose that we want to offer clients "fault-tolerant request execution"
  - We can replace a traditional service with a group of members
  - Each request is assigned to a primary (ideally, spread the work around) and a backup
    - Primary sends a "cc" of the response to the request to the backup
  - Backup keeps a copy of the request and steps in only if the primary crashes before replying
- Sometimes called "coordinator/cohort" just to distinguish from "primary/backup"

Publish / Subscribe

- Goal is to support a simple API:
  - Publish(“topic”, message)
  - Subscribe(“topic”, event_hander)
- We can just create a group for each topic
  - Publish multicasts to the group
  - Subscribers are the members

Scalability warnings!

- Many existing group communication systems don’t scale incredibly well
  - E.g. JGroups, Ensemble, Spread
  - Group sizes limited to perhaps 50-75 members
  - And individual processes limited to joining perhaps 50-75 groups (Spread: see next slide)
- Overheads soar as these sizes increase
  - Each group runs protocols oblivious of the others, and this creates huge inefficiency

Publish / Subscribe issue?

- We could have thousands of topics!
  - Too many to directly map topics to groups
- Instead map topics to a *smaller set* of groups.
  - SPREAD system calls these “lightweight” groups
  - Mapping will result in inaccuracies... Filter incoming messages to discard any not actually destined to the receiver process
- Cornell’s new QuickSilver system will instead directly support immense numbers of groups

Other “toolkit” ideas

- We could embed group communication into a framework in a “transparent” way
  - Example: CORBA fault-tolerance specification does lock-step replication of deterministic components
  - The client simply can’t see failures
    - But the determinism assumption is painful, and users have been unenthusiastic
    - And exposed to correlated crashes

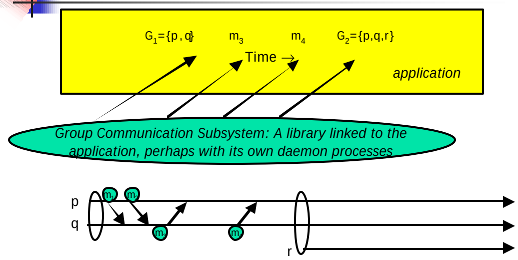
Other similar ideas

- There was some work on embedding groups into programming languages
  - But many applications want to use them to link programs coded in different languages and systems
  - Hence an interesting curiosity but just a curiosity
- More work is needed on the whole issue

Existing toolkits: challenges

- Tensions between threading and ordering
  - We need concurrency (threads) for perf.
  - Yet we need to preserve the order in which “events” are delivered
- This poses a difficult balance for the developers

Preserving order



The tradeoff

- If we deliver these upcalls in separate threads, concurrency increases but order could be lost
- If we deliver them as a list of event, application receives events in order but if it uses thread pools (think SEDA), the order is lost

Solution used in Horus

- This system
  - Delivered upcalls using an event model
  - Each event was numbered
  - User was free to
    - Run a single-threaded app
    - Use a SEDA model
- Toolkit included an "enter/leave region in order" synchronization primitive
  - Forced threads to enter in event-number order

Other toolkit "issues"

- Does the toolkit distinguish members of a group from clients of that group?
  - In Isis system, a ***client*** of a group was able to multicast to it, with vsync properties
  - But only ***members*** received events
- Does the system offer properties "across group boundaries"?
  - For example, using cbcast in multiple groups

Features of major virtual synchrony platforms

- Isis: First and no longer widely used
  - But was perhaps the most successful; has major roles in NYSE, Swiss Exchange, French Air Traffic Control system (two major subsystems of it), US AEGIS Naval warship
  - Also was first to offer a publish-subscribe interface that mapped topics to groups

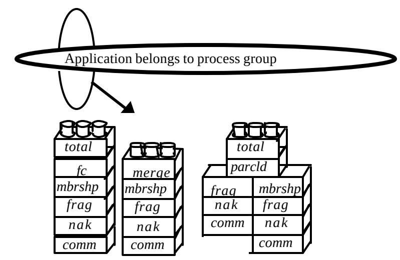
Features of major virtual synchrony platforms

- Totem and Transis
  - Sibling projects, shortly after Isis
  - Totem (UCSB) went on to become Eternal and was the basis of the CORBA fault-tolerance standard
  - Transis (Hebrew University) became a specialist in tolerating partitioning failures, then explored link between vsync and FLP

Features of major virtual synchrony platforms

- Horus, JGroups and Ensemble
  - All were developed at Cornell: successors to Isis
  - These focus on flexible protocol stack linked directly into application address space
    - A stack is a pile of micro-protocols
    - Can assemble an optimized solution fitted to specific needs of the application by plugging together "properties this application requires", lego-style
    - The system is optimized to reduce overheads of this compositional style of protocol stack
  - JGroups is _very_ popular.
  - Ensemble is somewhat popular and supported by a user community. Horus works well but is not widely used.

Horus/JGroups/Ensemble protocol stacks



JGroups (part of JBoss)

- Developed by Bela Ban
  - Implements group multicast tools
    - Virtual synchrony was on their "to do" list
    - But they have group views, multicast, weaker forms of reliability
  - Impressive performance!
  - Very popular for Java community
- Downloads from www.JGroups.org

Spread Toolkit

- Developed at John Hopkins
  - Focused on a sort of "RISC" approach
    - Very simple architecture and system
    - Fairly fast, easy to use, rather popular
  - Supports one large group within which user sees many small "lightweight" subgroups that seem to be free-standing
  - Protocols implemented by Spread "agents" that relay messages to apps

Summary?

- Role of a toolkit is to package commonly used, popular functionality into simple API and programming model
- Group communication systems have been more popular when offered in toolkits
  - If groups are embedded into programming languages, we limit interoperability
  - If groups are used to transparently replicate deterministic objects, we're too inflexible
- Many modern systems let you match the protocol to your application's requirements