CS5412: TRANSACTIONS (II)

Lecture XVIII

Ken Birman

Today's topic

- How do cloud systems actually use transactions?
- Last time we saw the basic transactional model.
  - But as we saw from reviewing Brewer's CAP theorem and the BASE methodology, transactions are sometimes too expensive and not scalable enough
- This has led to innovations on the transaction side
  - Snapshot isolation (related to serializability and ACID)
  - Business transactions (related to BASE)

Snapshot Isolation

- This idea started with discussion about lock-based (pessimistic) concurrency control in comparison with timestamp-based concurrency control
  - With locking we incur high costs to obtain one lock at a time. In distributed settings these costs are prohibitive.
    - Deadlock is a risk, must use a deadlock avoidance scheme
  - With timestamped concurrency control, we just pick a time at which transactions will run.
    - If times are picked to be unique, progress guaranteed because some transaction will have the smallest TS and won't abort. But others may abort and be forced to retry

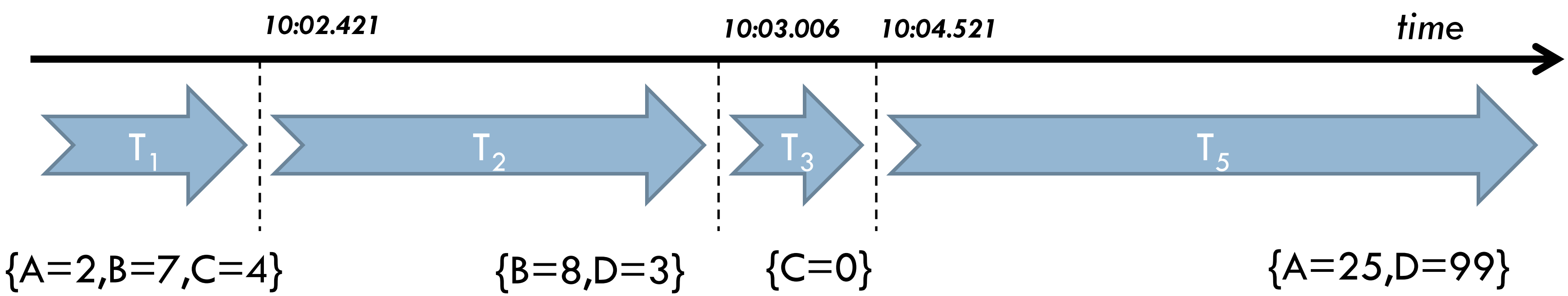
Pros and cons

- Each scheme attracted a following
  - Locking is easy to design and works well if transactions do a great deal of updates/writes
  - But 2PC can be costly if transactions are doing mostly reads and few writes

  - In contrast, timestamp schemes work very well for read-mostly or pure-read workloads and do a lot of rollback if a workload has a mixture

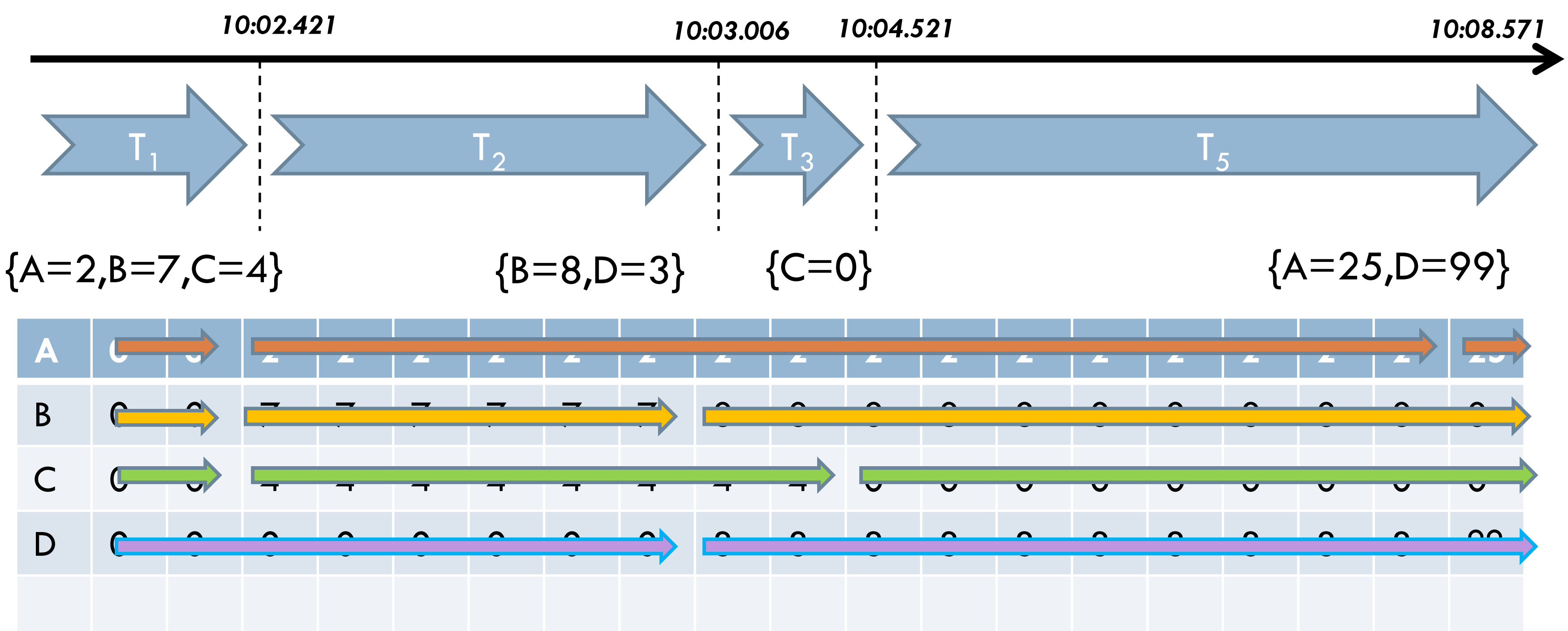
Snapshot isolation

- Arose from database products that offered "multiversion" data
  - Popular in the cloud, because we sometimes don't want to throw anything away
  - Each transaction can be seen as moving the database from a consistent state to a new consistent state



A multiversion database

- Instead of just keeping the value of the variables in the database, we track each revision and when the change was committed



Snapshot isolation idea

- For a read transaction, just pick a time at which the reads should be executed (ideally, a recent time corresponding to the commit of some transaction)
  - If transactions really take us from consistent state to consistent state, this will be a "safe" time to execute
  - Reads don't change the state so execute without risk of needing to abort
- Then use locking to execute transactions that need to perform update operations

Fancier snapshot isolation

- Often used for *all* reads, not just read-only transactions
- Runs dynamically: Instead of picking just one time at which to run, pick a “range” of times and track it
- A single window is used even if X accesses many variables

Fancier snapshot isolation

- … pick a "range" of times and track it
  - E.g. transaction X might initially pick time range [0…NOW]
  - As X actually accesses variables, narrow the time window of the transaction [max(old start, new start), min(old end, new end)]
    - E.g. X tries to read variable A and because A is locked for update by transaction Y, reads A=2
    - A=2 was valid from time [10:02.421,10:08.57]
    - This narrows the window of validity for transaction X

How can a window vanish?

- Occurs if there just isn't any point in the serialization order at which this set of reads could have happened
- Result of an update that invalidates some past read
- Causes transaction to abort

Complications

- In fact, snapshot isolation doesn't guarantee full serializability
  - An update transaction might "invalidate" a read by updating A at an unexpectedly early time
  - Unless we check the read-only transactions won't know which ones to abort
  - Real issue: X may already have finished
- If we use s.o. for reads in read/write transactions, we get additional "bad cases"

Snapshot isolation is widely used

- Works well with multitier cloud computing infrastructures
  - Caching structures that track validity intervals for cached variables are common
  - Several papers have shown how to make snapshot isolation fully serializable, but methods haven't been widely adopted (and may never be)
- Fits nicely with BASE: Basically available, soft state replication with eventual consistency
  - Often we don't worry about consistency for the client

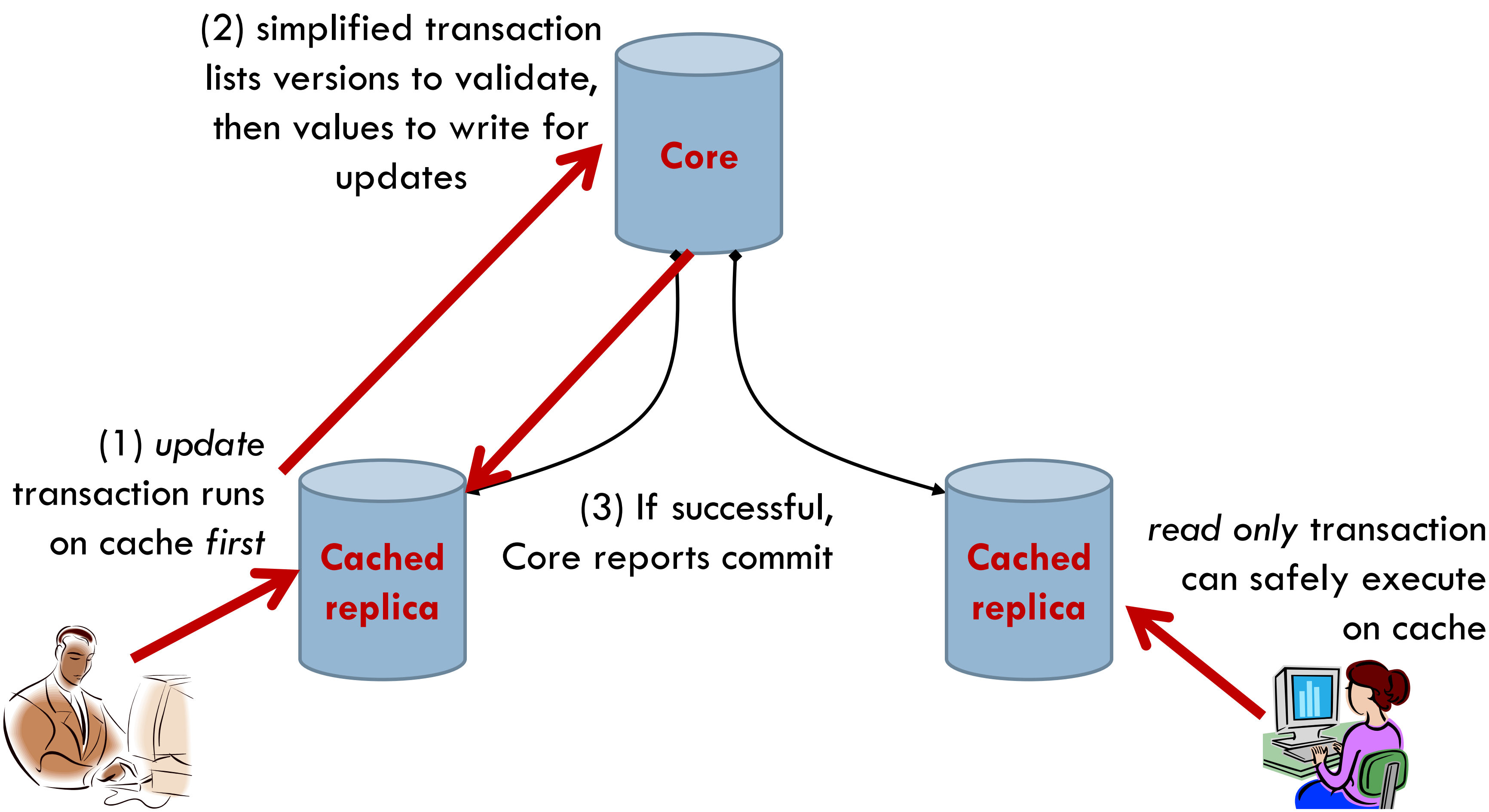
Consistency: Two "views"

- Client sees a snapshot of the database that is internally consistent and "might" be valid
- Internally, database is genuinely serializable, but the states clients saw aren't tracked and might sometimes become invalidated by an update
- Inconsistency is tolerated because it yields such big speedups, although some clients see "wrong" results

Do clients need perfect truth?

- If so, one recent idea is to "validate" at commit time
  - Many systems have a core transactional system that does updates
  - Collections of read-only cached replicas are created at the edge where clients reside
  - Read-only transactions run on these (true) replicas, with no risk of error
  - Read/write transactions track the versions read and the changes they "want" to make (intentions list)
- Then package these intended changes as ultra-fast transactions to be sent to the core system
  - It checks that these versions are still current,and if so, applies the updates, like in the Sinfonia system (discussed in class)
  - If not, transaction "aborts" and must be retried
- Effect is to soak up as much hard work as possible at the edge

A picture of how this works



Core issue: How much contention?

- Root challenge is to understand
  - How many updates will occur
  - How often those updates conflict with concurrent reads or with concurrent updates

- In most of today's really massive cloud applications either contention is very rare, in which case transactional database solutions work, or we end up cutting corners and relaxing consistency

Tradeoff: Scale versus consistency

- With a core system we can impose strong consistency, but doing so limits scalability
  - It needs to "validate" every update
  - At some point it will get overloaded

- But if we don't use a core system we can't guarantee consistency
  - We may be able to design the application to tolerate small inconsistencies. Many web systems work this way

Are there other options?

- How does this approach compare with scalable replication using Paxos or Virtual Synchrony?

- In those systems the "contention" related to the order in which multicasts were delivered
  - Virtual synchrony strives to find ways of weakening required ordering to gain performance
  - Paxos is like serializability: One size fits all. But this is precisely why Brewer ended up proposing CAP!

Business transactions

- The Web Services standards introduces (yet) another innovation in the space
- They define a standard transactional API for cloud computing, and this is widely supported by transactional products of all kinds
- But they also define what are called "business transactions"

Think of Expedia

- You book a trip to Costa Rica
  - Flight down involves two separate carriers
  - Fourteen nights in a total of three hotels
  - Rental car for six days, bus tours for the rest
  - Two rainforest tours, one with “zip line experience”
  - Dinner reservation for two on your friend’s birthday at the Inka Grill restaurant in San Jose
  - Travel insurance covering stomach ailiments (costs extra)
  - Special “babysit your dog” service in Ithaca

Should this be one transaction?

- Traditionally the transactional community would have argued that cases like these are precisely what transactions were invented for

- In practice… it makes little sense to use transactions
  - Multiple services, perhaps with very distinct APIs (e.g. may just need to phone the Inka Grill directly)
  - Many ways to roll back if something goes wrong, like just cancelling the car reservation

Concept of a business transaction

- Instead of a single transaction, models something like this as a whole series of separate transactions
  - Maybe in a few cases done as true transactions
  - But others might be done in business-specific ways

- The standard assumes that each has its own specialized rollback technology available

- It also requires a "reliable message queuing" system

Reliable message queuing

- Basically, email for programs
  - Like with normal email, can send messages to addresses and they will be held until read/deleted
  - Spooler is assumed to be highly available and reliable
  - Generally has some kind of multi-stage structure: spools messages near the sender until handed off to the server, and only deleted once safely logged

How this works

- Application “sends” a set of requests, like one email each
- Spooler accepts the set and executes them one by one, restarting any that are disrupted by crashes
- Handling of other kinds of failures (“Sorry sir, the restaurant is fully booked that night”) is under programmatic control
  - You need to add details to tell the system what to do
  - It won’t know that the Mexicali Cafe is a fallback

Business transactions

- We create a sequence of transactions and of the associated undo actions for each
  - Spool the series of transactions, linked by a business-transaction-identifier
  - As each is executed, the undo action is spooled but in a "disabled" state
  - On commit of the final transaction in the sequence, the undo actions are deleted
  - On abort, the undo actions are enabled and run as a kind of reverse business transaction

Business transactions and BASE

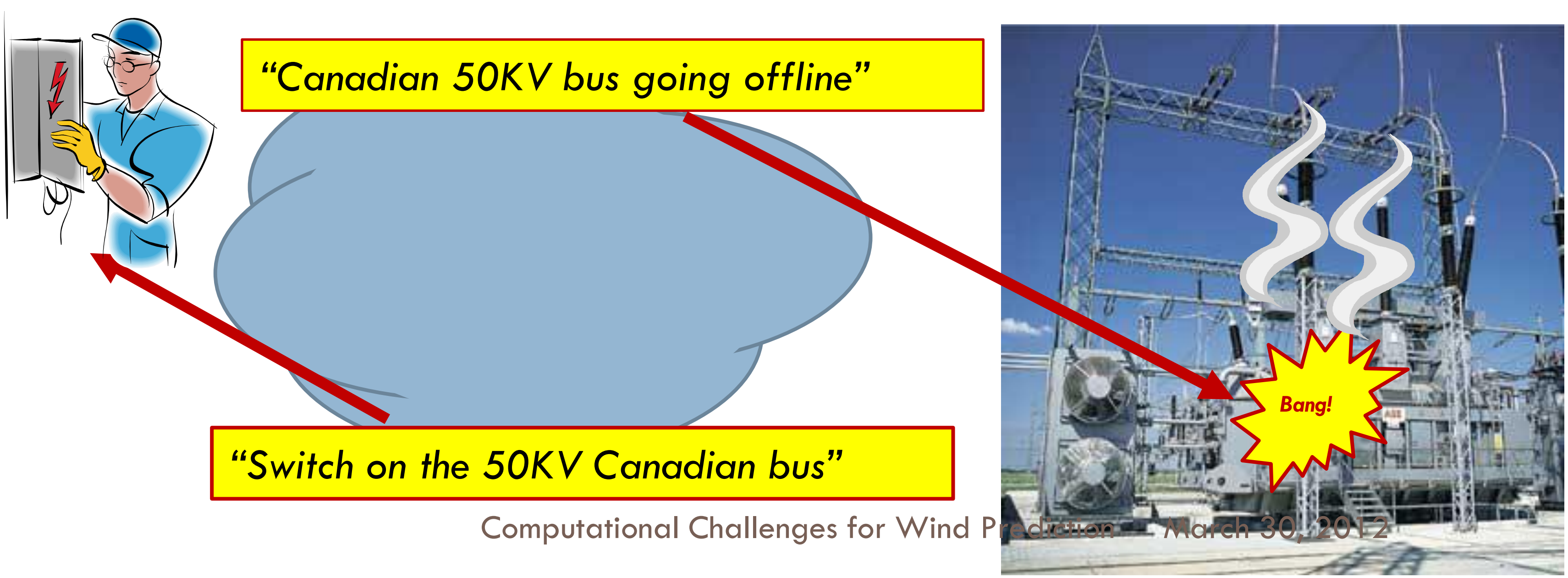
- If our reservations go part-way through but then the dog-sitter step fails, we end up leaving the world in a kind of inconsistent state
  - But soon after we run the undo actions and this reverses the problems we created
  - Even if someone failed to get a reservation at Inka Grill because of your temporarily booked table, they won't be so surprised when they try again in a few days and now a table is free

“Consistency is much overrated”

- We hear this a lot lately
- But you also need to wonder… what about
  - Medical care systems that run on the Internet?
  - Google’s self-driving cars?
  - The smart power grid

If eBay (BASE) ran the power grid

- With BASE, control system could have "two voices"
- In physical infrastructure settings, consequences can be very costly



The big problem

- Scalable consistency is hard!
  - Not impossible... but harder than weak consistency, or no consistency

- Today's most profitable web ventures manage quite well with weak models like BASE
  - Run a lot of stuff in parallel
  - Replicate data when you get a chance, but no rush
  - Sweep any errors under the rug

What happens tomorrow?

- Nobody can compete with the cloud "price point"
  - In modern technology, the cheapest solution always wins
  - It becomes the only option available
  - So everything migrates to the winner
- We've seen this again and again
- The cloud will win. You guys will build the winning solutions, and they will be cloud based!

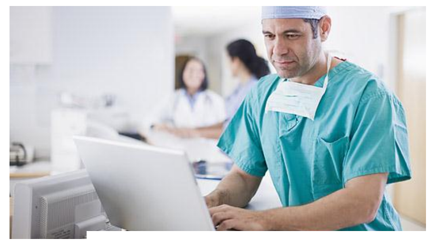
Why is it hard to cloudify high assurance?

- Let's look at Isis^2
- A cloud-based high assurance story…
- Can we view it as a blueprint for cloud-scale resiliency of a kind the masses might adopt?

High assurance: Different perspectives

- A single platform has many kinds of "users"

Programmer: Depends on platform properties but treats implementation as a black box.



End user: Seeks confidence that the system is safe and that if it goes offline, a warning will appear

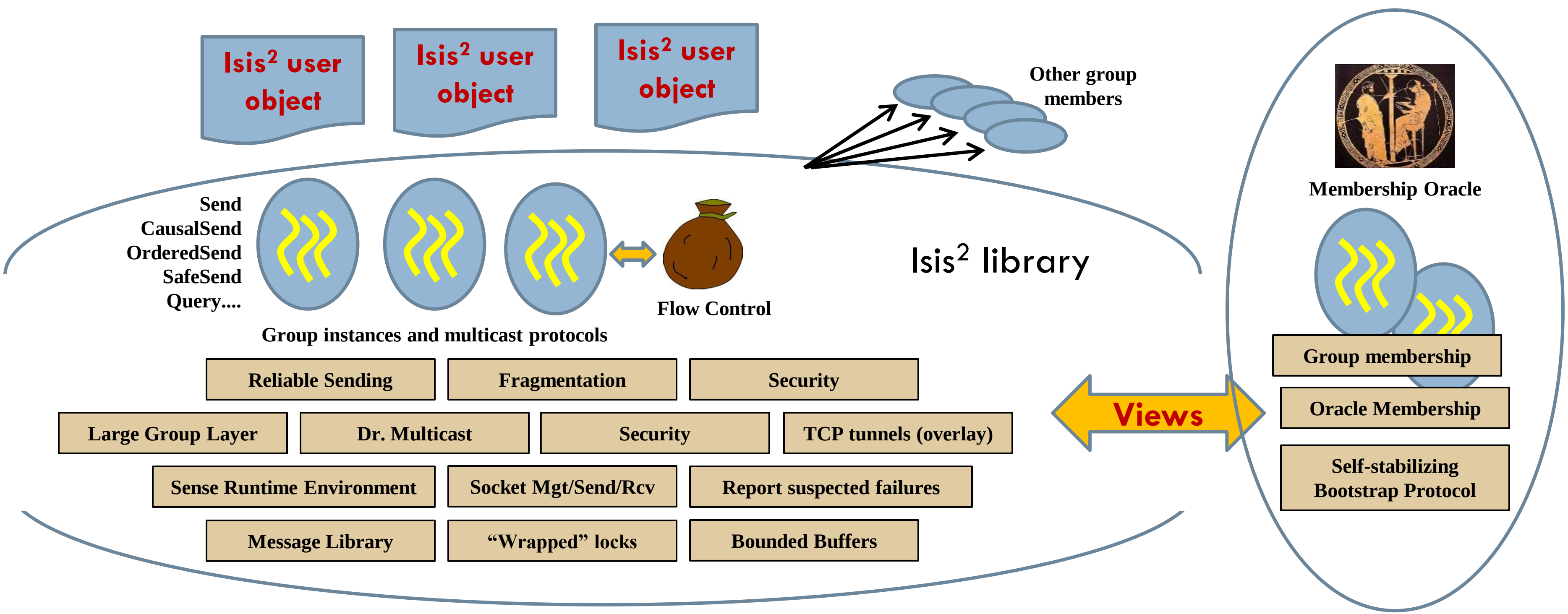
Datacenter operator: Requires scalability, elasticity, and guarantees that applications won't disrupt shared resources

Protocol designer: Uses formal specification and logic to prove implementation of protocols correct.

- Each brings different objectives and requires different methods

It takes a "community"

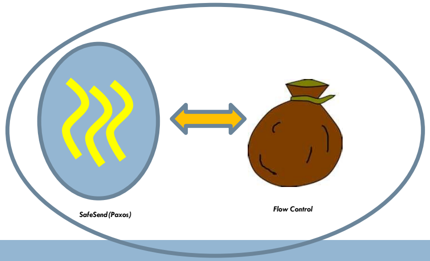
- Formal methods tempt us to reason about a single instance of a single protocol at a time:
  - "Paxos with *n* members = { x, y, z, … } and α acceptors…"
- Yet real systems are complex and concurrent with many interacting component parts that must operate in concert



Layers approach properties in distinct ways

- The lower layers of Isis^2 focus on stochastic assurances aimed at the datacenter owner
  - For example, the system won't abuse IP multicast and will employ a good flow control method
- Protocol layers focus on temporal logic properties
  - For example, the Paxos guarantees of agreement on events, their order, and of durability for decided events
- User focuses on a simplified abstraction
  - Virtual synchrony multicast within groups
- End-user relies upon properties achieved by the application, but doesn't worry about how it was built

Consider flow control

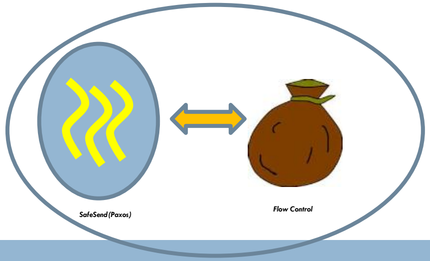


- Consider SafeSend (Paxos) within Isis[2]
  - Basic protocol looks very elegant
  - Not so different from Robbert’s 60 lines of Erlang

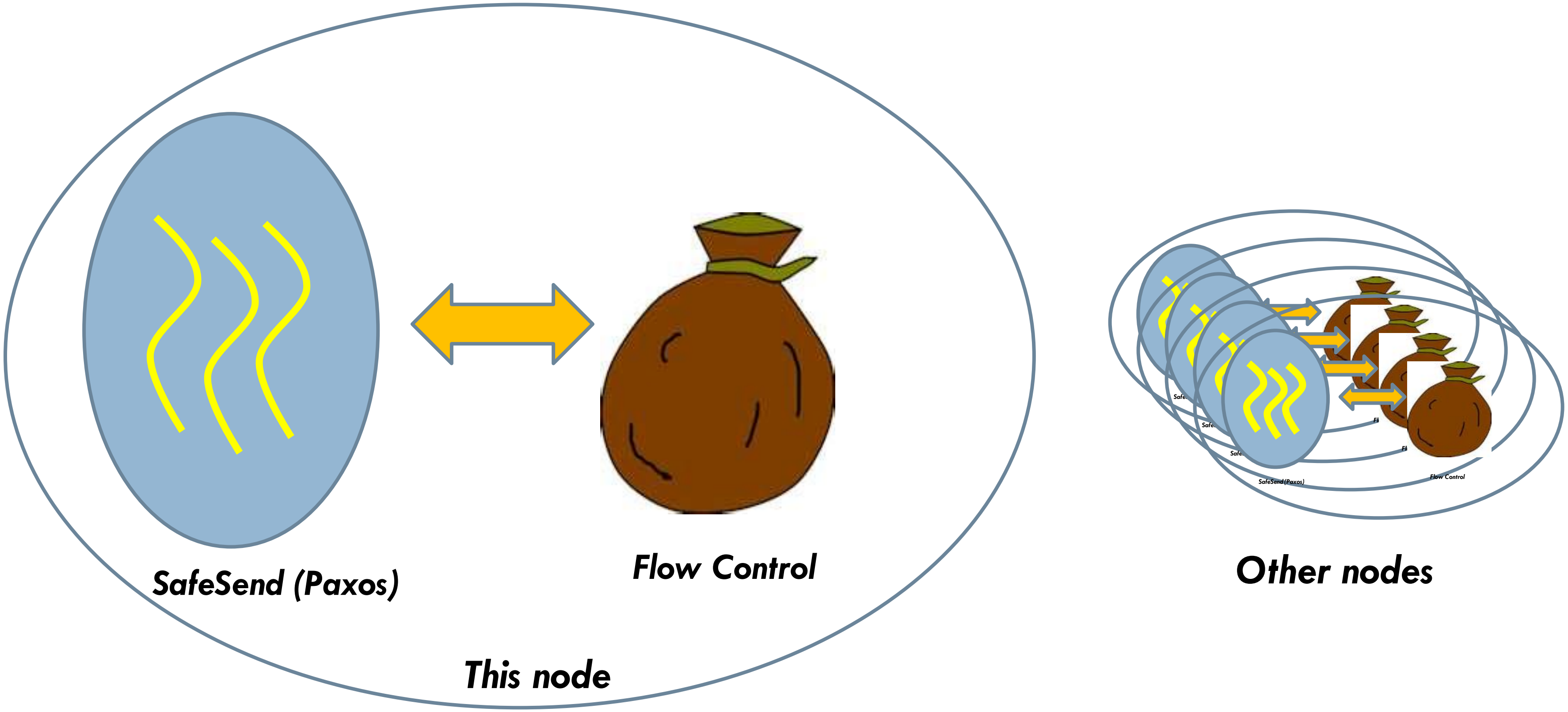
- But pragmatic details clutter this elegant solution
  - E.g.: Need “permission to send” from flow-control module
  - … later tell flow-control that we’ve finished

- Flow control is needed to prevent overload
  - Illustrates a sense in which Paxos is “underspecified”

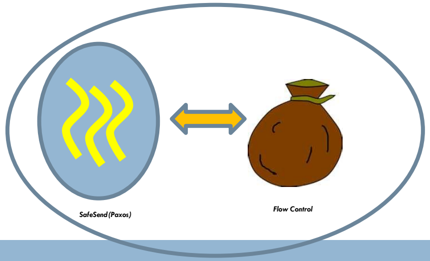
Pictoral representation



- "Paxos" state depends on "flow control state"
- Modules are concurrent. "State" spans whole group



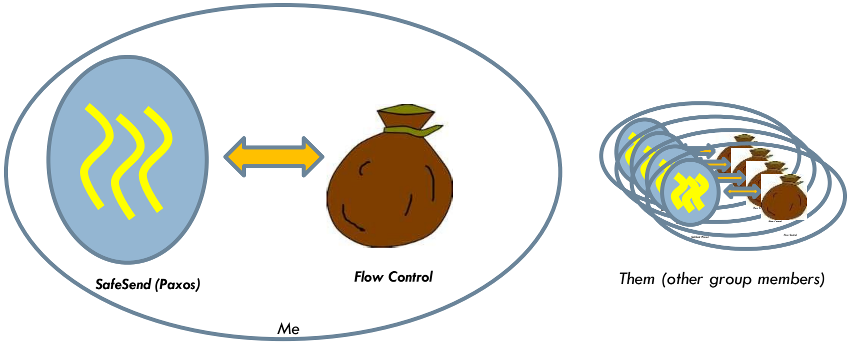
… flow control isn't local



- One often thinks of flow control as if the task is a local one: "don't send if my backlog is large"
- But actual requirement turns out to be distributed
  - "Don't send if the *system as a whole* is congested"
    - Permission to initiate a SafeSend obtains a "token" representing a unit of backlog at this process
    - Completed SafeSend must return the token
    - Flow Control module tracks backlog states of full set of group members, hence needs a rule for reporting state via multicast
    - Must also monitor group membership and unblock senders if a failure "frees" enough backlog to enable senders to resume
  - Thus Flow Control is a non-trivial distributed protocol!

This creates a new challenge

- Previously, could have proved Paxos safe+live in the virtual synchrony model
  - Virtual synchrony views play the role of a failure detector (an eventually strong one, in the sense of ◊ S)
  - Paxos lives in a simpler world and can be proved fully correct
- But now we see that Paxos would be "dependent" upon the flow control module, and vice versa!
  - Paxos needs permission to send
  - Flow control needs to track protocols in progress
  - Group members need to track each-other's states

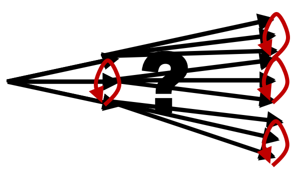


Paxos + Flow Control correctness?

- Flow control imposed only when a protocol starts
  - Waiting for flow control induces a partial dependency ordering
  - If prior protocols are live, some waiting protocol will eventually have a chance to run
  - Fairness requires further mechanisms…

It isn't quite so simple: Delay Tolerance

- Recall that Isis^2 targets cloud-scale settings
  - Hence aggressively scaled, must "ride out" scheduling delays, long message latencies, elasticity events
  - Most work on DTNs focuses on progress "despite" delays
  - But in Isis^2 if some nodes get far ahead of other nodes, the flow-control module we've just discussed kicks in! This defeats DTN logic
- Given this mix of needs, which the best 2PC implementation?
  - One leader, *n* members
  - Hierarchical (tree)
  - Tree of rings (Ostrowski: QSM)
  - Hypothetical: Self-stabilization or gossip "emulation" of 2PC



- ... And whichever we favor also needs to lend itself to an implementation we can prove correct!

Lessons one learns... and challenges

- Formal models are powerful conceptual tools
  - Impossible to build a system like $Isis^2$ without them
  - And $Isis^2$ in turn enables high-assurance applications

- Yet our science of formal methods remains too narrow in its focus
  - Teaches us how to reason about a single protocol
  - But also need to think about communities of protocols, concurrency everywhere, cross-process dependencies

The challenge?

- Which road leads forward?
  1. Extend our formal execution model to cover all elements of the desired solution: a "formal system"
  2. Develop new formal tools for dealing with complexities of systems built as communities of models
  3. Explore completely new kinds of formal models that might let us step entirely out of the box

The challenge?

- Which road leads forward?
  1. Extend our formal execution model to cover all elements of the desired solution: a "formal system"

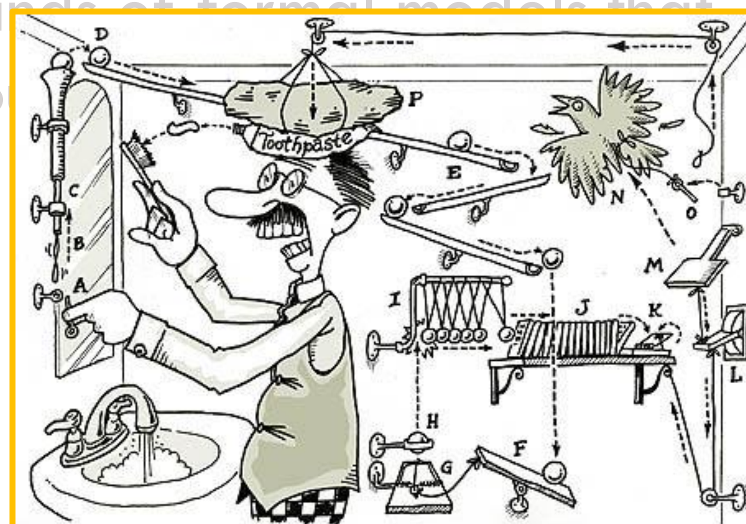
Doubtful:

- **The resulting formal model would be unwieldy**
- **Theorem proving obligations rise more than linearly in model size**

els

3. Explore completely new ki

might let us step entirely o



The challenge?

- Which road leads forward?
  1. Extend our formal execution model to cover all elements of the desired solution: a "formal system"
  2. Develop new formal tools for dealing with complexities of systems built as communities of models
  3. Explore completely new kinds of formal models that

Possible, but hard:

- **Need to abstract behaviors of these complex "modules"**
- **On the other hand, this is how one debugs platforms like Isis^{2}**

The challenge?

- Which road leads forward?

1. Extend our formal execution model to cover all elements of the desired solution: a "formal system"
2. Develop new formal tools for dealing with complexities of systems built as communities of models
3. Explore completely new kinds of formal models that might let us step entirely out of the box

Intriguing:

- **All of this was predicated on a style of deterministic, agreement-based model**
- **Could self-stabilizing protocols be composed in ways that permit us to tackle equally complex applications but in an inherently simpler manner?**

Summary

- We've seen several high assurance "stories"
  - Paxos
  - Virtual synchrony
  - Transactions
- In each case the cloud community says "too expensive" and even proves theorems like CAP
  - But while "just say no" is easy, results are sometimes harmful.
  - Must we accept a low-assurance cloud?
- And yet things that need high assurance are coming