CS5412: HOW MUCH ORDERING?

Lecture XVI

Ken Birman

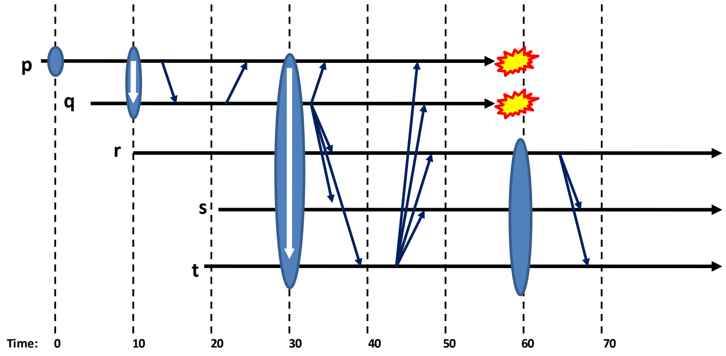
Ordering

- The key to consistency turns has turned out to be delivery ordering (durability is a "separate" thing)
  - Given replicas that are ***initially*** in the same state…
  - … if we apply the same updates (with no gaps or dups) in the same order, they ***stay*** in the same state.

- We've seen how the virtual synchrony model uses this notion of order for
  - Delivering membership view events
  - Delivery of new update events

But what does “same order” mean?

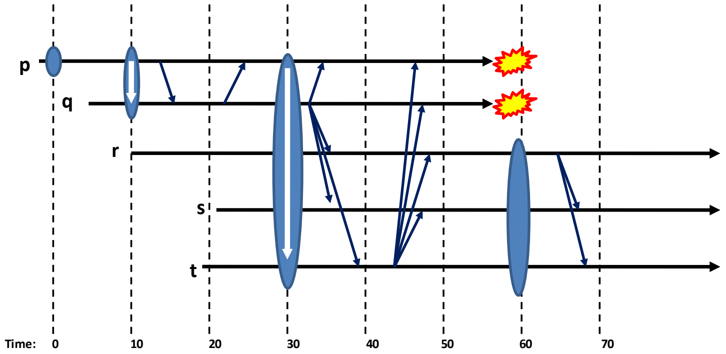
- The easy answer is to assume that the “same order” means just what is says
  - Every member gets every message in the identical sequence
  - This was what we called a “synchronous” behavior
- Better term might be “closely” synchronous since we aren’t using synchronous clocks



Synchronous execution

As an example…

- Suppose some group manages variables X and Y
- P sends updates to X and Y, and so does Q
  - P: X = X-2
  - Q: X = 17.3
  - Q: Y = Y*2 + X
  - T: Y = 99



- The updates "conflict": order matters
- The model keeps the replicas synchronized

But what if items have “leaders”

- Suppose all the updates to X are by P
- All the updates to Y are by Q
- Nobody ever looks at X and Y “simultaneously”

- Could this ever arise?
  - Certainly! Many systems have **leaders**
  - Consider the electric power grid: when monitoring sensors, there is often one process connected to a given sensor and only that process multicasts the updates

Does this impact ordering?

- Now the rule is simpler

- As long as we perform updates in the order the leader issued them, for each given item, the replicas of the item remain consistent

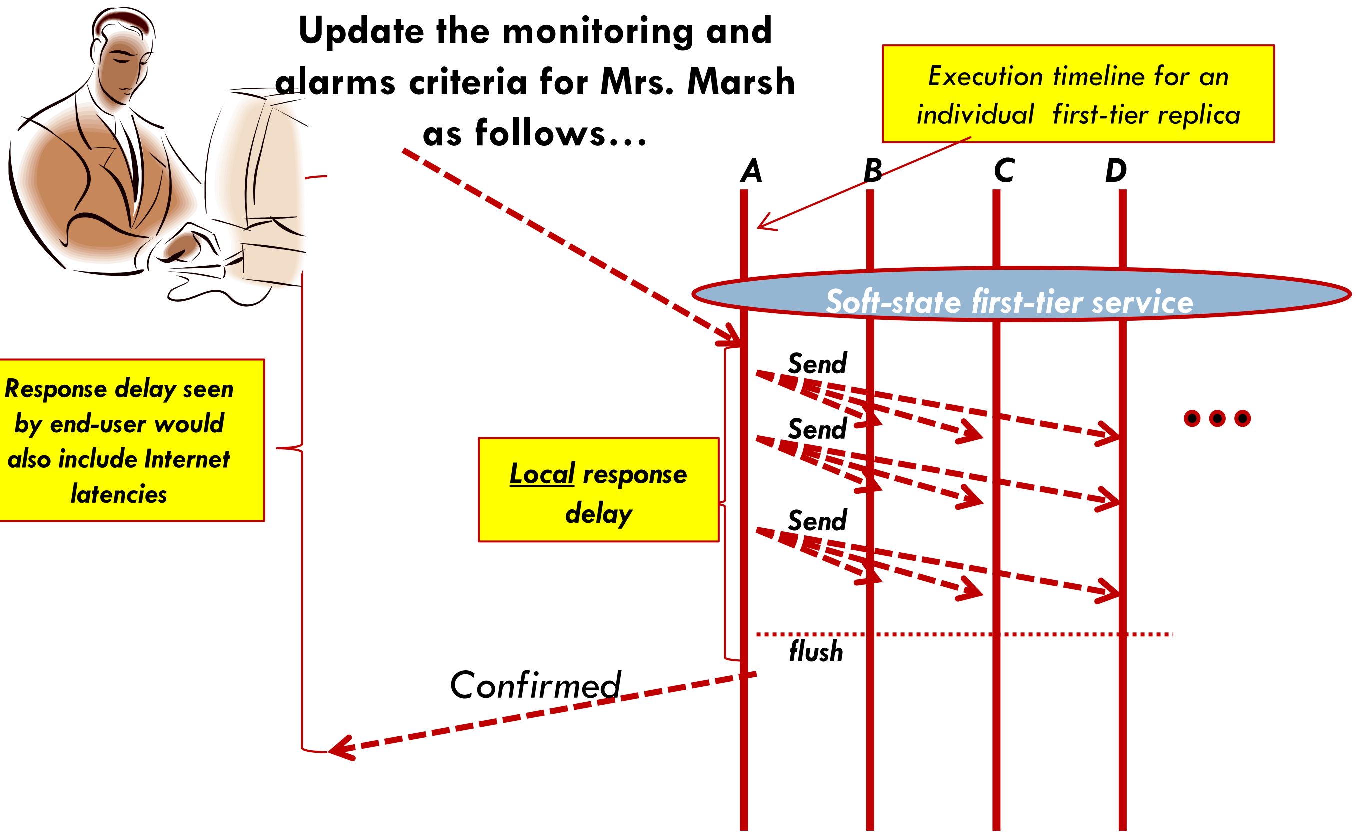
- Here we see a "FIFO" ordering: with multiple leaders we have multiple FIFO streams, but each one is behaving "like" a 1-n version of TCP

What does FIFO mean for messages?

- In other situations, FIFO means "first in, first out"

- For a message system, we mean "keep the sending order used by the process that sent the messages"
  - With multiple senders, receivers could see different orderings on incoming messages. If they update the same objects inconsistency arises.
  - But with a single sender for each kind of update, and no conflicting actions by different senders, FIFO suffices

Revisiting our medical scenario

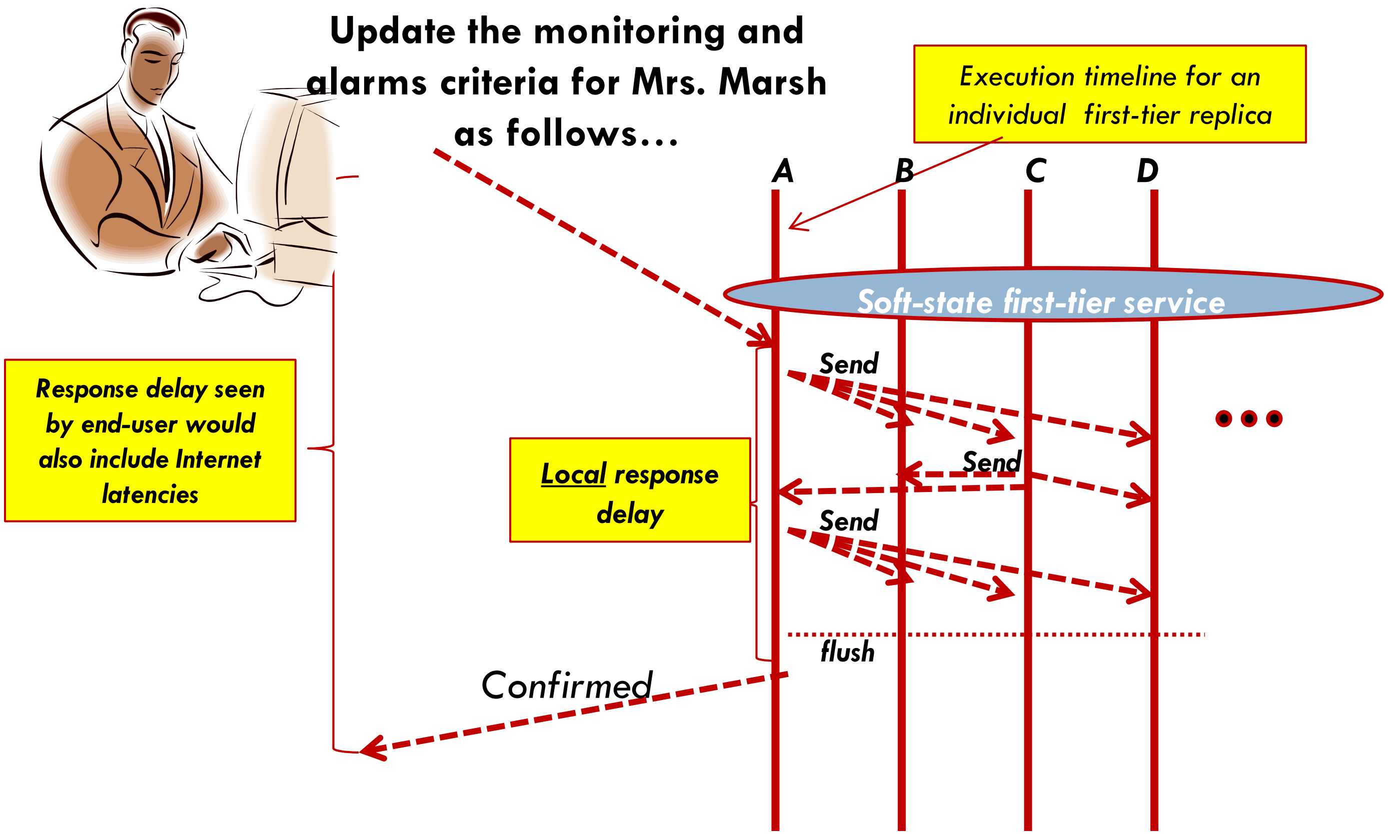


- If A is the only process to handle updates, a FIFO Send is all we need to maintain consistency

From FIFO to causal…

- A fancier FIFO ordering policy can also arise
- Consider P and Q that both update X but with locks:
  - First P obtains the lock before starting to do updates
  - Then it sends updates for item X for a while
  - Then it releases the lock and Q acquires it
  - Then Q sends updates on X, too
- What ordering rule is needed here?

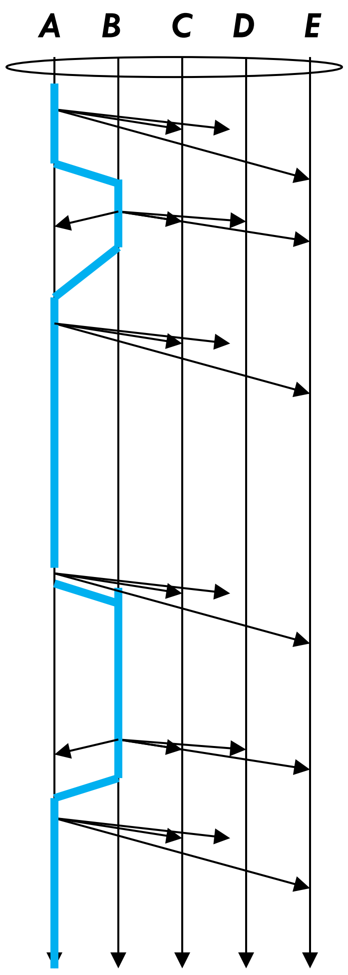
Causal ordering "variation"



- Notice that the send by C is "after" the send by A

Mutual exclusion

- Dark blue when holding the lock
- Lock moving around is like a thread of control that moves from process to process
- Our goal is “FIFO along the causal thread” and the causal order is thus *exactly* what we need to enforce
- In effect, causal order is like total order except that the sender “moves around” over time



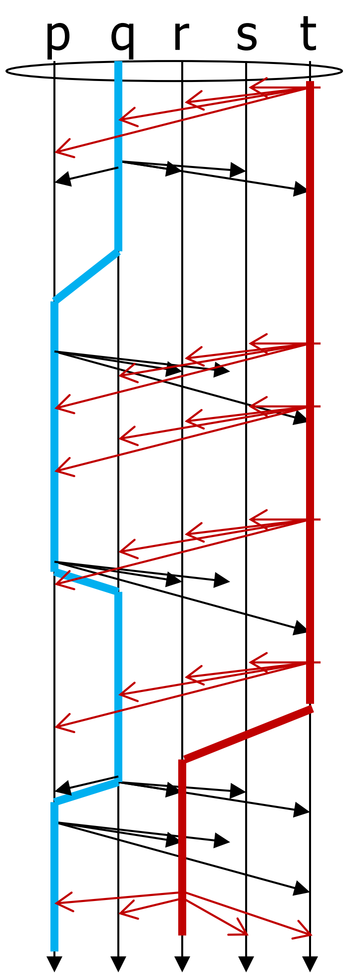
Causal ordering

- This example touches on a concept Leslie Lamport calls "causal ordering"
  - A's release of the lock on X to B "caused" B to issue updates on X. When B was done, A resumed.
  - The update order is A's, then B's, then A's.

- Lamport's happened-before relation captures this
  - If P sends **m**, and Q sends **m'**, and **m** $\rightarrow$ **m'**, then we want **m** delivered before **m'**
  - Called a "causal delivery" rule

Same idea with several locks

- Suppose red defines the lock on X
- Blue is the lock on Y
- The "relative" ordering of X/Y updates isn't important because those events commute: they update different variables
- Causal order captures this too



Can we implement causal delivery?

- Think about how one implements FIFO multicast
  - We just put a counter value in each outgoing multicast
  - Nodes keep track and deliver in sequence order
- Substitute a vector timestamp
  - We put a *list of counters on each outgoing multicast*
  - Nodes deliver multicasts only if they are *next in the causal ordering*
  - No extra rounds required, just a bit of extra space (one counter for each possible sender)

Total ordering

- Multicasts in a single agreed order no matter who sends them, without locking required. In $Isis^2$ the OrderedSend protocol guarantees this.

- SafeSend (Paxos) has this property

- OrderedSend: total ordering with <u>optimistic</u> delivery.
- SaeSend: total ordering with <u>pessimistic</u> delivery. This is slow but offers the strongest form of durability

Levels of ordering one can use

- No ordering or even no reliability (like IP multicast)
- FIFO ordering (requires an integer counter)
- Causal ordering (requires vector timestamps)
- Total ordering (requires a form of lock). Can be implemented as a "causal and total" order
- Paxos agreed ordering (tied to strong durability)

- Isis^2 offers *all* of these options

Consistent cuts and Total Order

- Recall our discussion of consistent cuts
  - Like an "instant in time" for a distributed system
  - Guess what: An event triggered by a totally ordered message delivery ***happens on a consistent cut!***
  - For example, it is safe to use a totally ordered query to check for a deadlock, or to count something
    - The answer will be "correct"
    - No ghost deadlocks or double counting or undercounting

Isis^2 multicast primitives

Names for Primitives

- **RawSend:** No guarantees
- **Send:** FIFO
- **CausalSend:** Causal order
- **OrderedSend:** Total order
- **SafeSend:** Paxos

Additional Options

- Flush: Durability (not needed for SafeSend)
- In-memory/disk durability (SafeSend only)
- Ability to specify the number of acceptors (SafeSend)

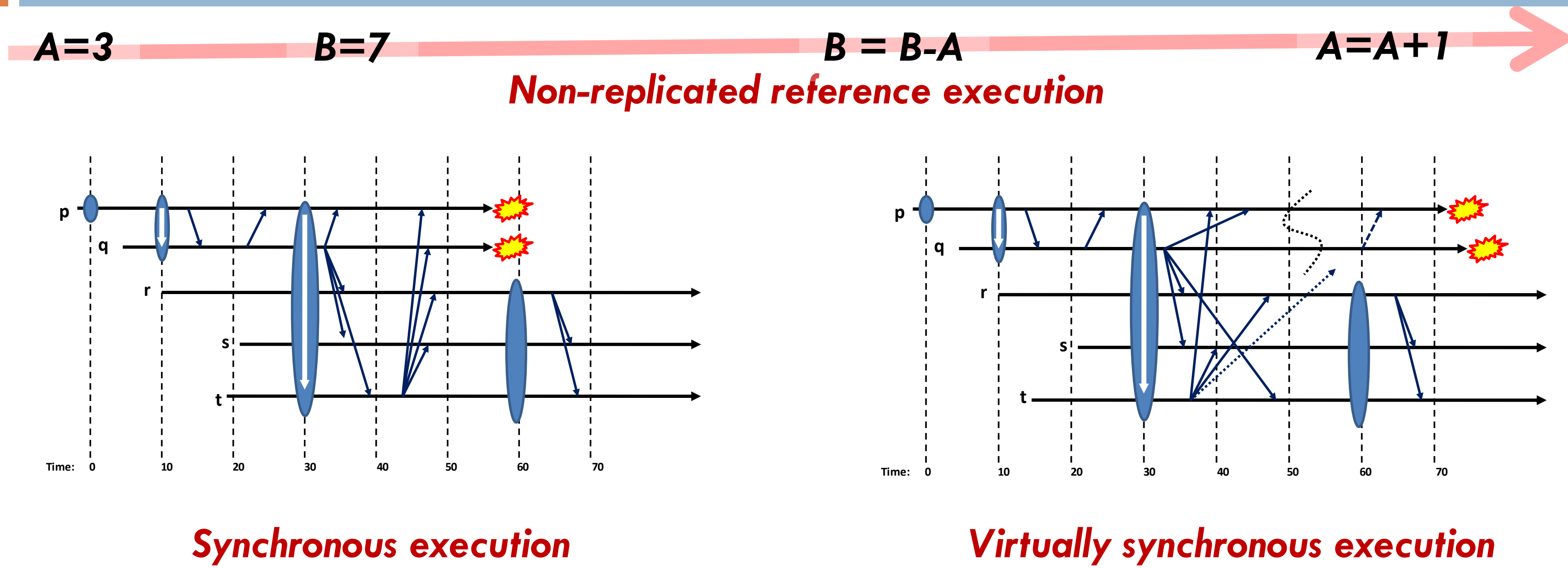
… all come in P2P and multicast forms, and all can be used as basis of Query requests

Will people need so many choices?

- Most developers start by using
  - OrderedSend for situations where strong durability isn't a key requirement (total order)
  - SafeSend if total order plus strong durability is needed

- Then they switch to weaker ordering primitives if
  - Application has a structure that permits it
  - Performance benefit outweighs the added complexity
  - Using the right primitive lets you pay for *exactly* what you need

Virtual synchrony recap



- Virtual synchrony is a "consistency" model:
  - *Synchronous runs: indistinguishable from non-replicated object that saw the same updates (like Paxos)*
  - *Virtually synchronous runs are indistinguishable from synchronous runs*

Some additional Isis^2 features

- State transfer and logging
- User registers a method that can checkpoint group state, and methods to load from checkpoint
- Isis^2 will move such a checkpoint to a new member, or store it into a file, at appropriate times

Security

- Based on 256-bit AES keys

- Two cases: Key for the entire system, and per-group keys.
  - System keys: used to sign messages (not encrypt!)
  - Per-group keys: all data sent on the network is encrypted first
  - But where do the keys themselves get stored?

Security

- One option is to keep the key material outside of Isis^2 in a standard certificate repository
  - Application would start up, fetch certificate, find keys inside, and hand them to Isis^2
  - This is the recommended approach

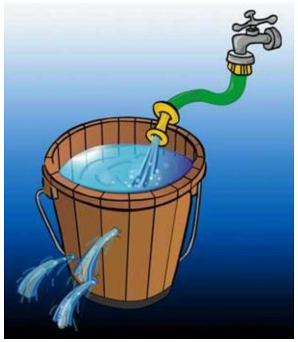
- A second option allows Isis^2 to create keys itself
  - But these will be stored *in files* under your user-id
  - File protection guards these: only you can access them
  - If someone were to log in as you, they could find the keys and decrypt group traffic

Flow control

- Two forms
- Built-in flow control is automatic and attempts to avoid overload situations in which senders swamp (some) receivers with too much traffic, causing them to fall behind and, eventually, to crash
- This is always in force except when using RawSend

Flow control

- The other form is user-controlled: You specify a "leaky bucket" policy, Isis^2 implements it
- Tokens flow into a bucket at a rate you can specify
- They also age out eventually (*leak*)
- Each multicast "costs" a token and waits if the bucket is empty
- Fully automated flow control appears to be very hard and may be impractical



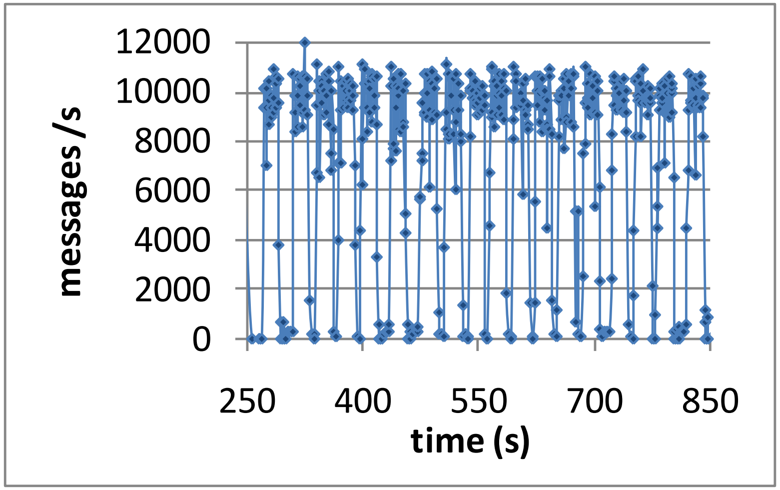
Dr. Multicast

- Something else Isis^2 does is to manage the choice of how multicast gets sent

- Several cases
  - Isis^2 can use IP multicast, if permitted. User controls the range of port numbers and the maximum number of groups
  - Isis^2 can send packets over UDP, if UDP is allowed and a particular group doesn't have permission to use Dr. Multicast
  - Isis^2 can "tunnel" over an overlay network of TCP links (a kind of tree with log(N) branching factor at each level)

Anatomy of a meltdown

- A “blend” of stories (eBay, Amazon, Yahoo):
  - Pub-sub message bus very popular. System scaled up. Rolled out a faster ethernet.
  - Product uses IPMC to accelerate sending
  - All goes well until one day, under heavy load, loss rates spike, triggering collapse
- Oscillation observed



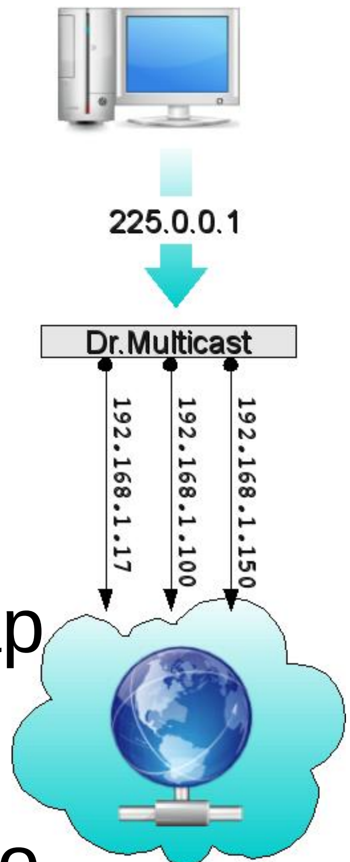
IPMC aggregation and flow control!

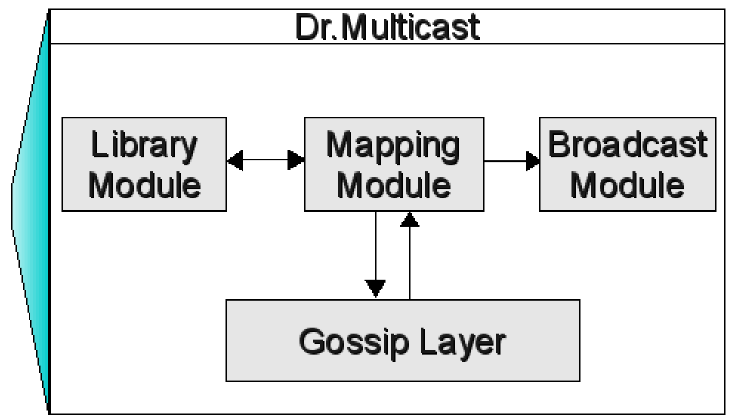
- Recall: IPMC became promiscuous because *too many multicast channels were used*
  - And this triggered meltdowns

- Why not *aggregate* (combine) IPMC channels?
  - When two channels have similar receiver sets, combine them into one channel
  - Filter (discard) unwanted extra messages

Dr. Multicast

- Application sees what looks like a normal IPMC interface (socket library)
- We intercept requests and map them to IPMC groups of our choice (or even to UDP)





Channel Aggregation

- Algorithm by Vigfusson, Tock
  papers: [HotNets 09, LADIS 2008]
- Uses a k-means clustering algorithm
  - Generalized problem is NP complete
  - But heuristic works well in practice

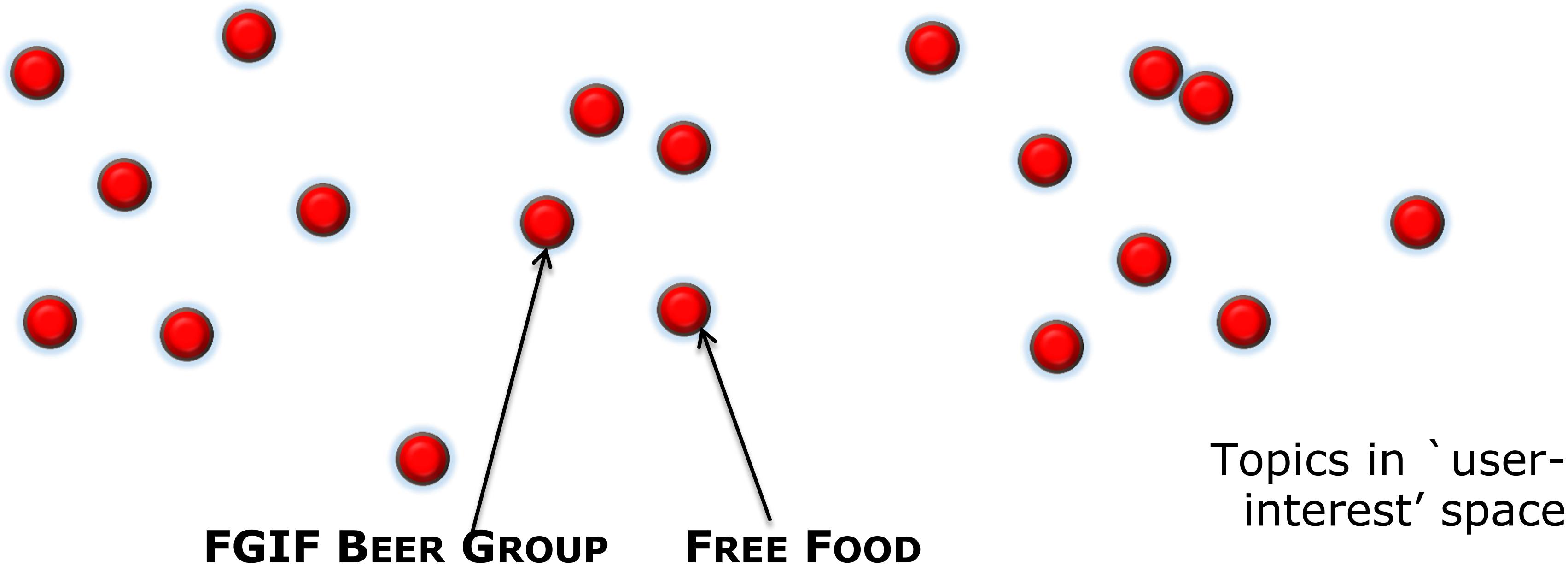
Optimization Questions

Dr. Multicast

- Assign IPMC and unicast addresses s.t.
  - $\leq \alpha$ % receiver filtering (hard)
  - (1) Min. network traffic
  - $\leq M$ # IPMC addresses (hard)

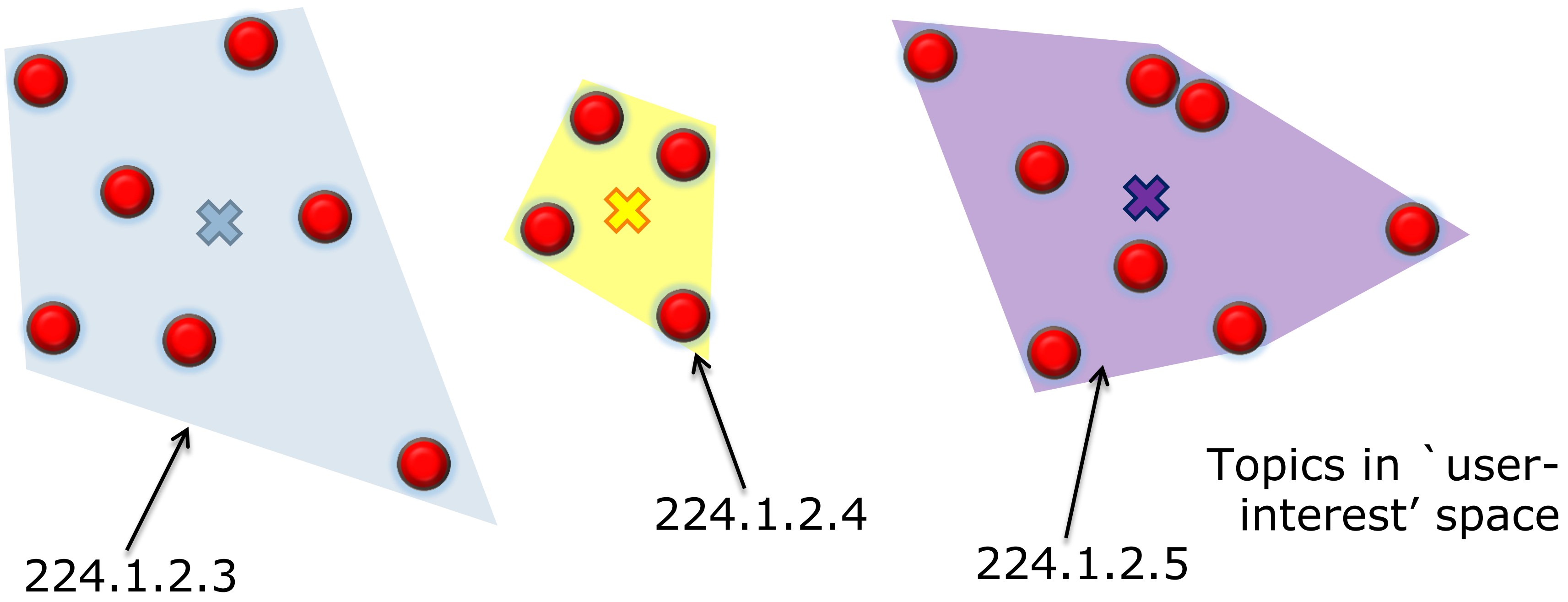
- Prefers sender load over receiver load
- Intuitive control knobs as part of the policy

MCMD Heuristic

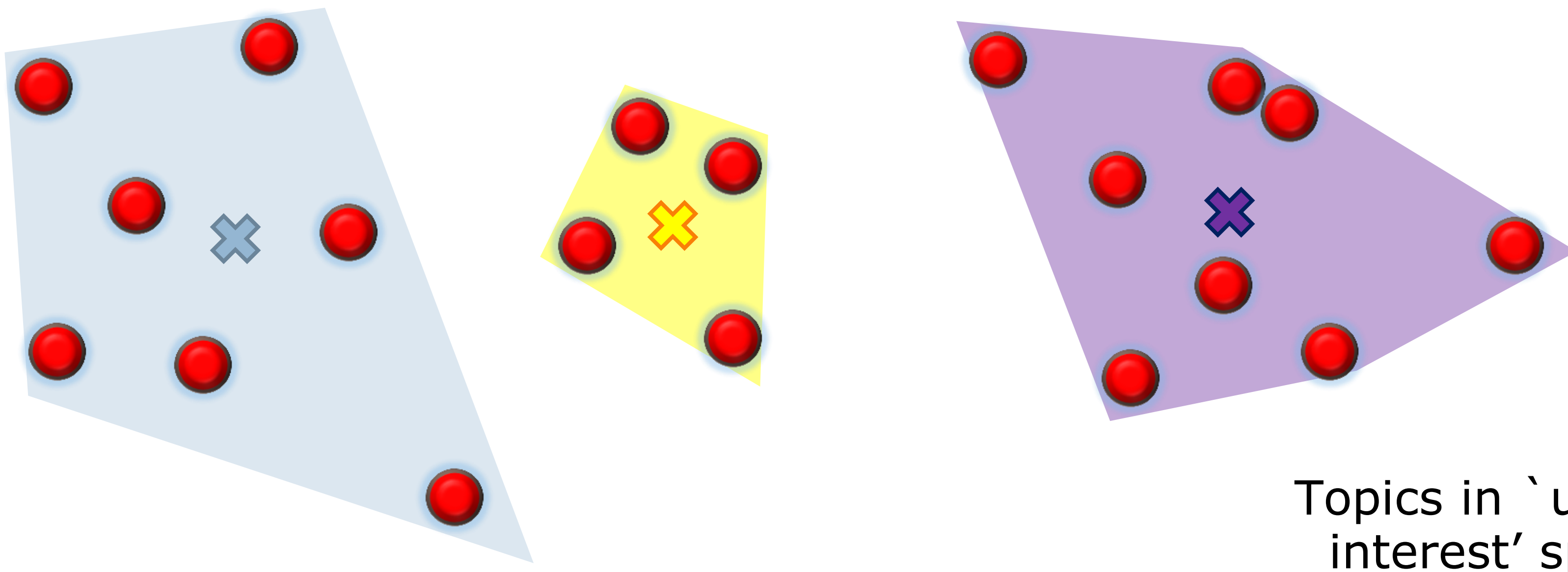


(0,1,1,1,1,0,1,0,0,0,1,1)

MCMD Heuristic



MCMD Heuristic

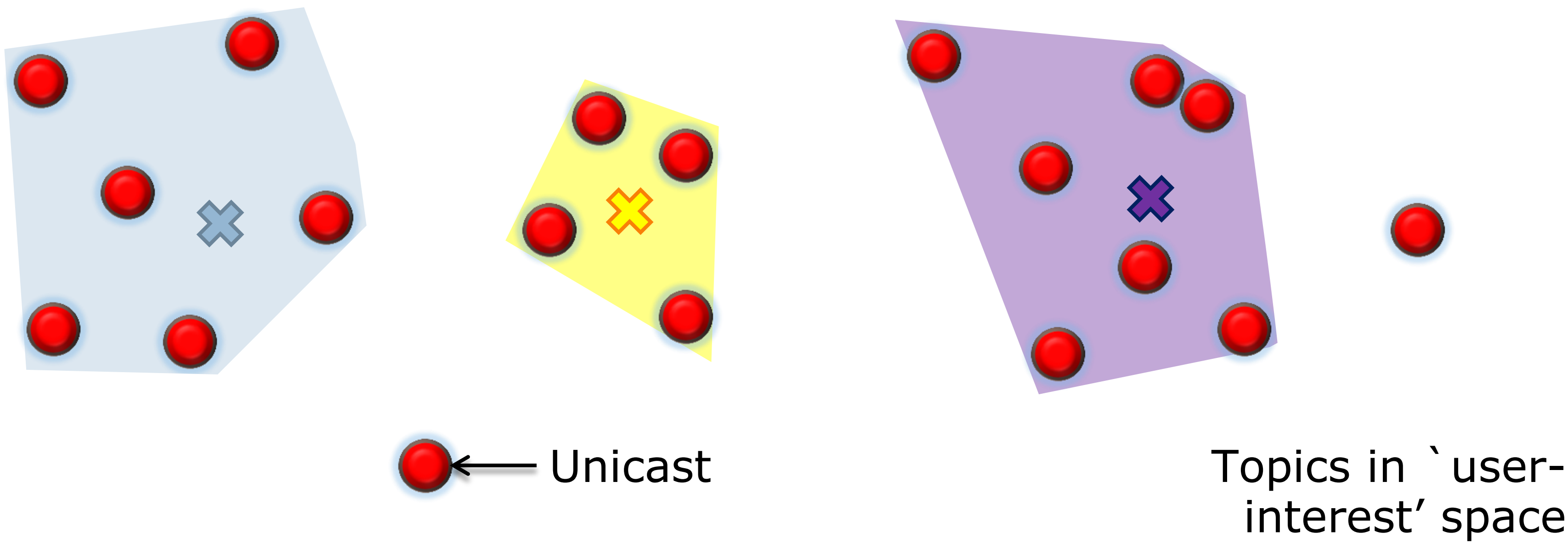


Topics in `user-interest' space

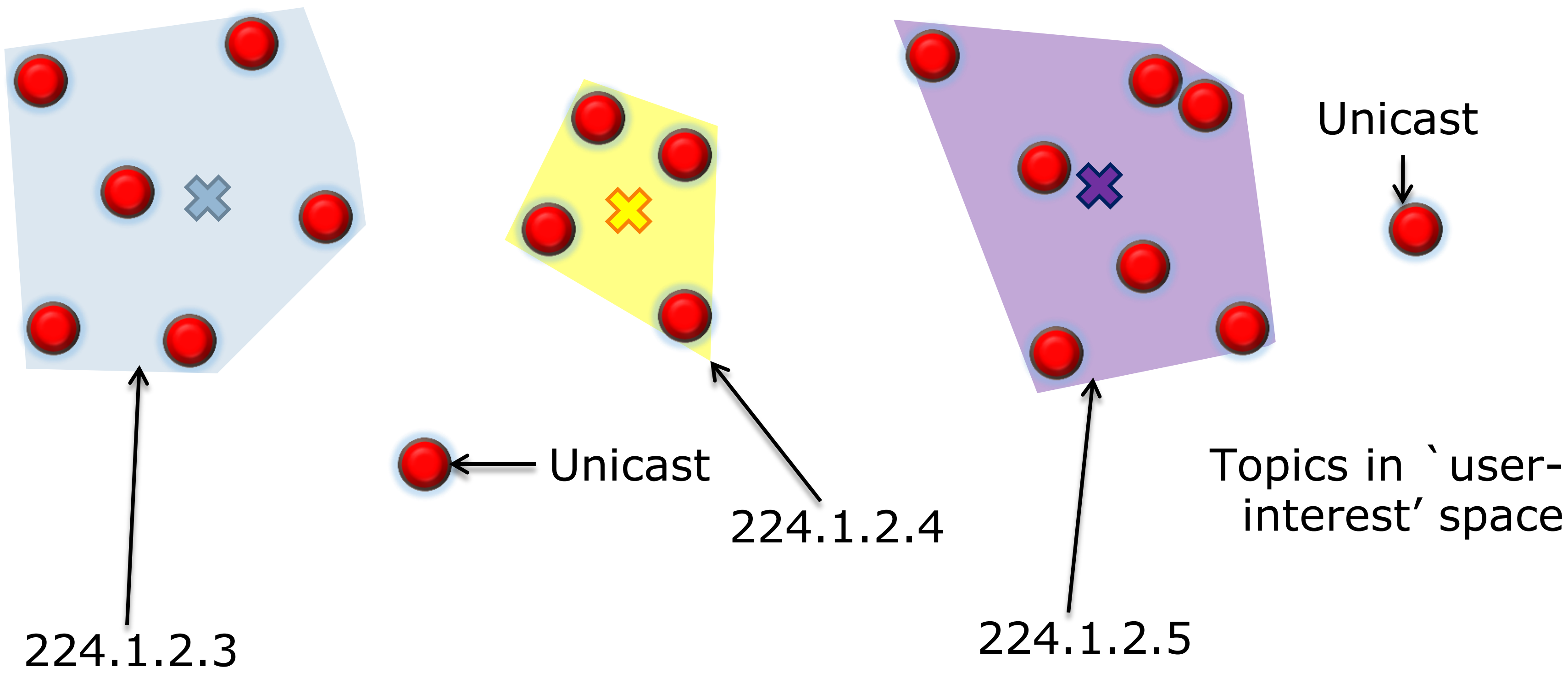
Sending cost:

Filtering cost:

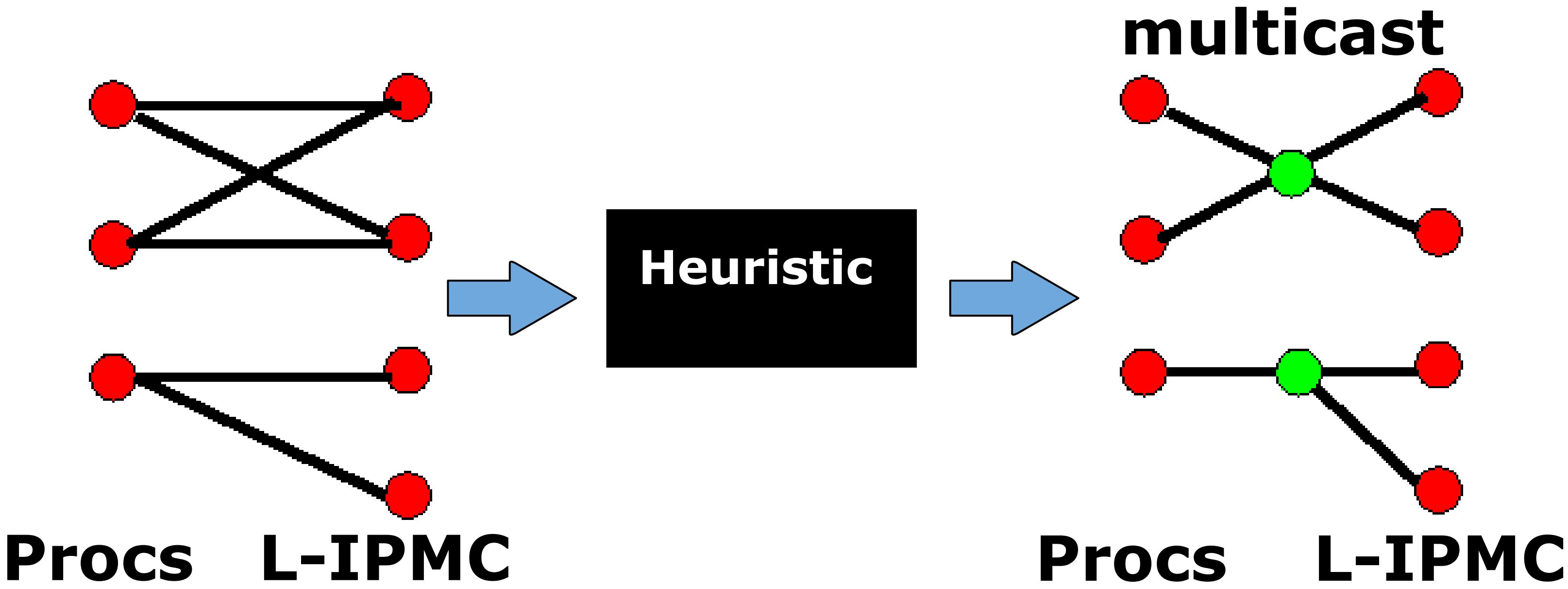
MCMD Heuristic



MCMD Heuristic



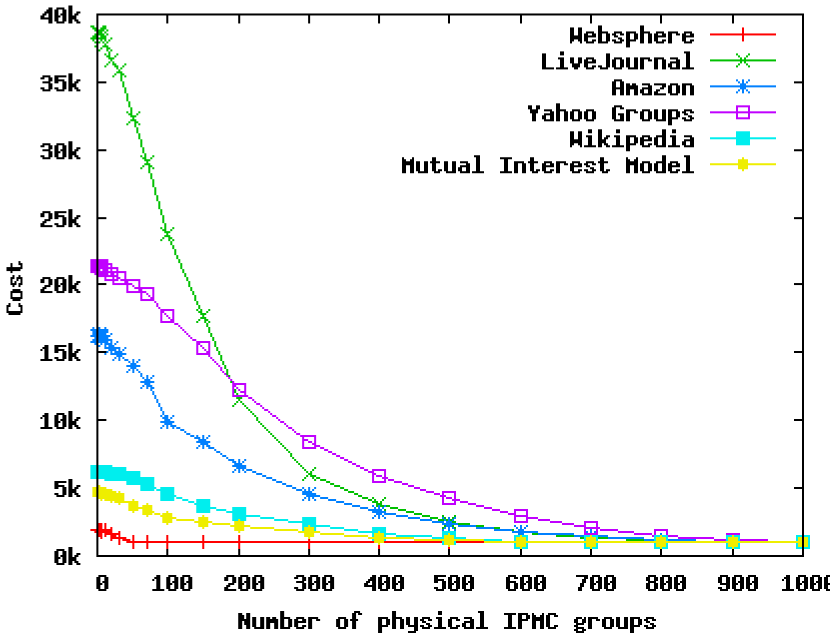
Using the Solution



- Processes use "logical" IPMC addresses
- Dr. Multicast transparently maps these to true IPMC addresses or 1:1 UDP sends

Effectiveness?

- We looked at various group scenarios
- Most of the traffic is carried by <20% of groups
- For IBM Websphere, Dr. Multicast achieves 18x reduction in physical IPMC addresses



- [**Dr. Multicast: Rx for Data Center Communication Scalability.** Ymir Vigfusson, Hussam Abu-Libdeh, Mahesh Balakrishnan, Ken Birman, and Yoav Tock. LADIS 2008. November 2008.]

Dr. Multicast in Isis^2

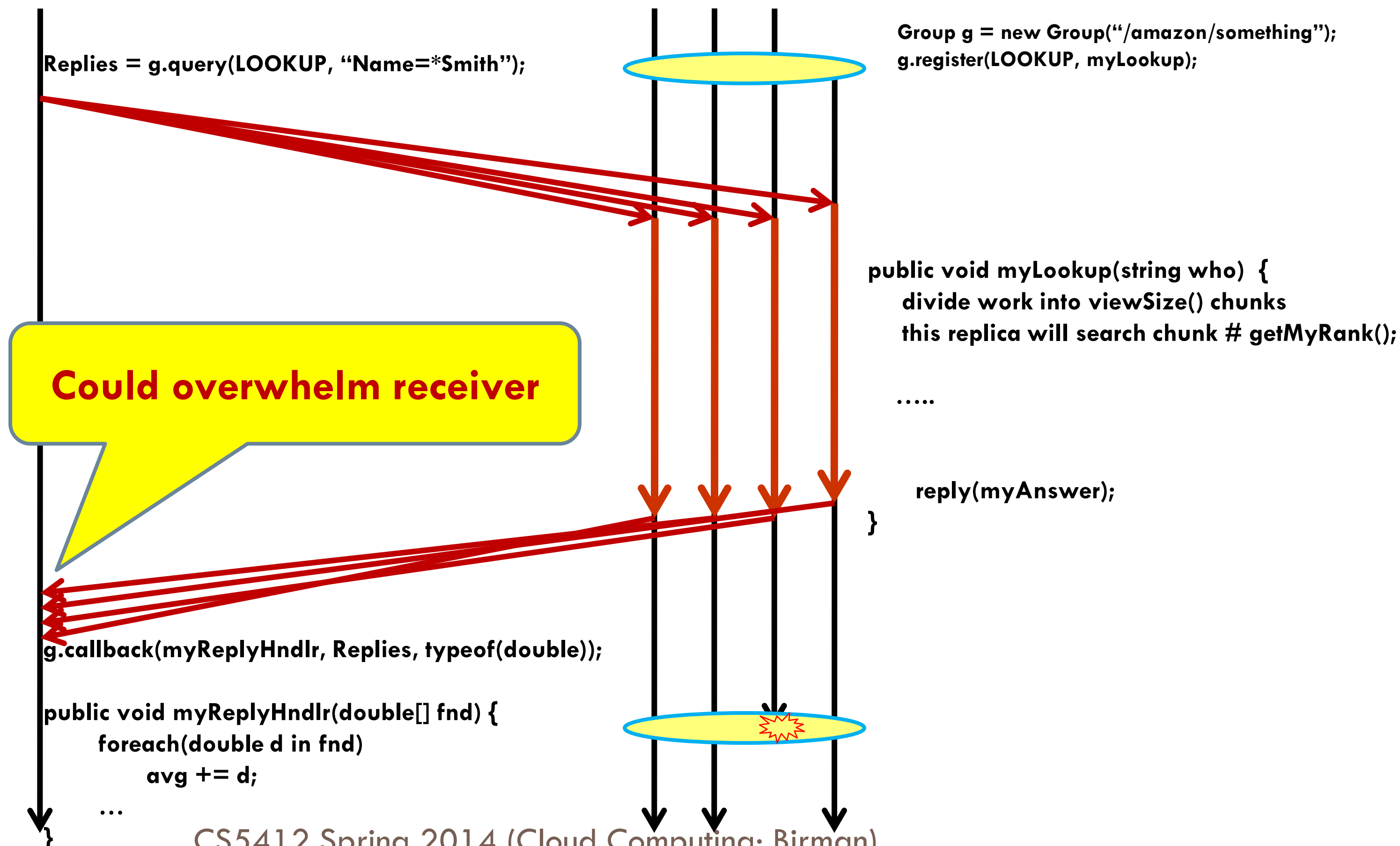
- System automatically tracks membership, data rates
- Periodically runs an optimization algorithm
  - Merges similar groups
  - Applies the Dr. Multicast greedy heuristic

- Isis^2 protocols "think" they are multicasting, but a logical to physical mapping will determine whether messages are sent via IPMC, 1-n UDP or the tree-tunnelling layer, all automatically

Large groups

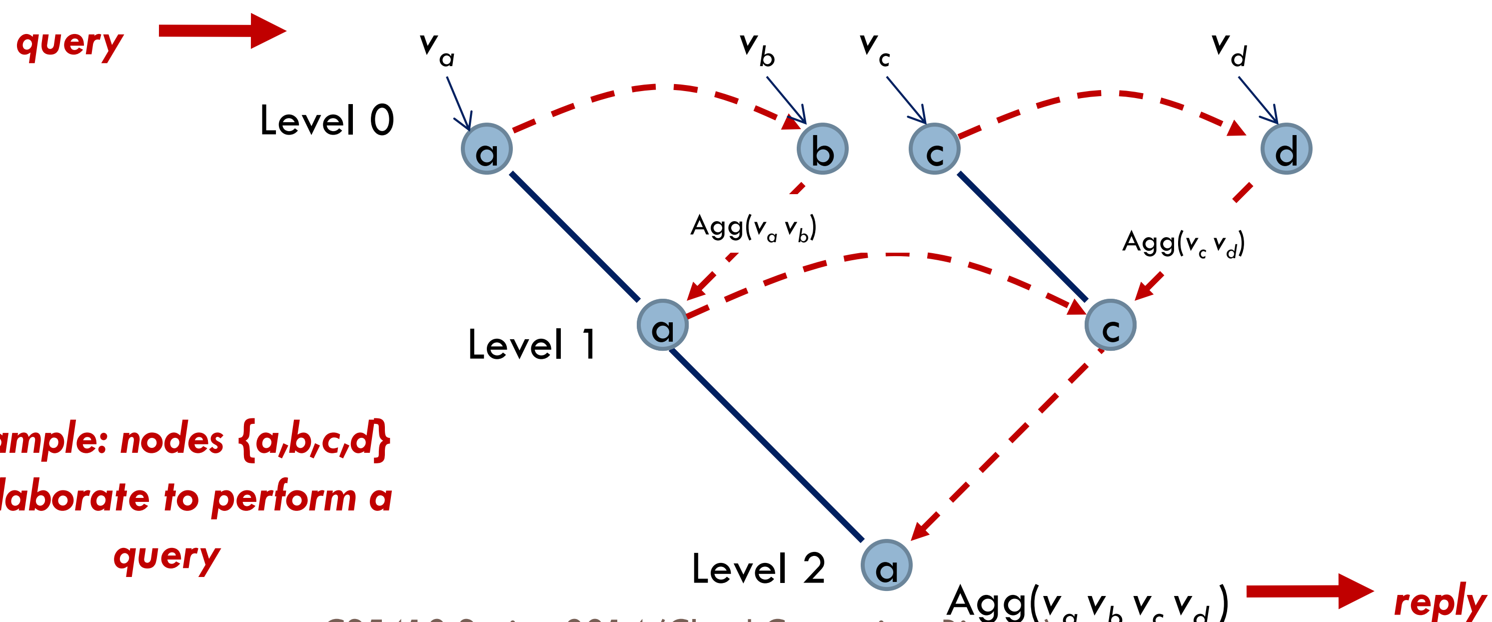
- Isis[2] has two styles of acknowledgment protocol
  - For "small" groups (up to ~1000 members), direct acks
  - Large groups use a tree of token rings: slower, but very steady (intended for 1000-100,000 members)
  - Also supports a scalable way to do queries with massive parallelism, based on "aggregation"
  - Very likely that as we gain experience, we'll refine the way large groups are handled

Example: Parallel search



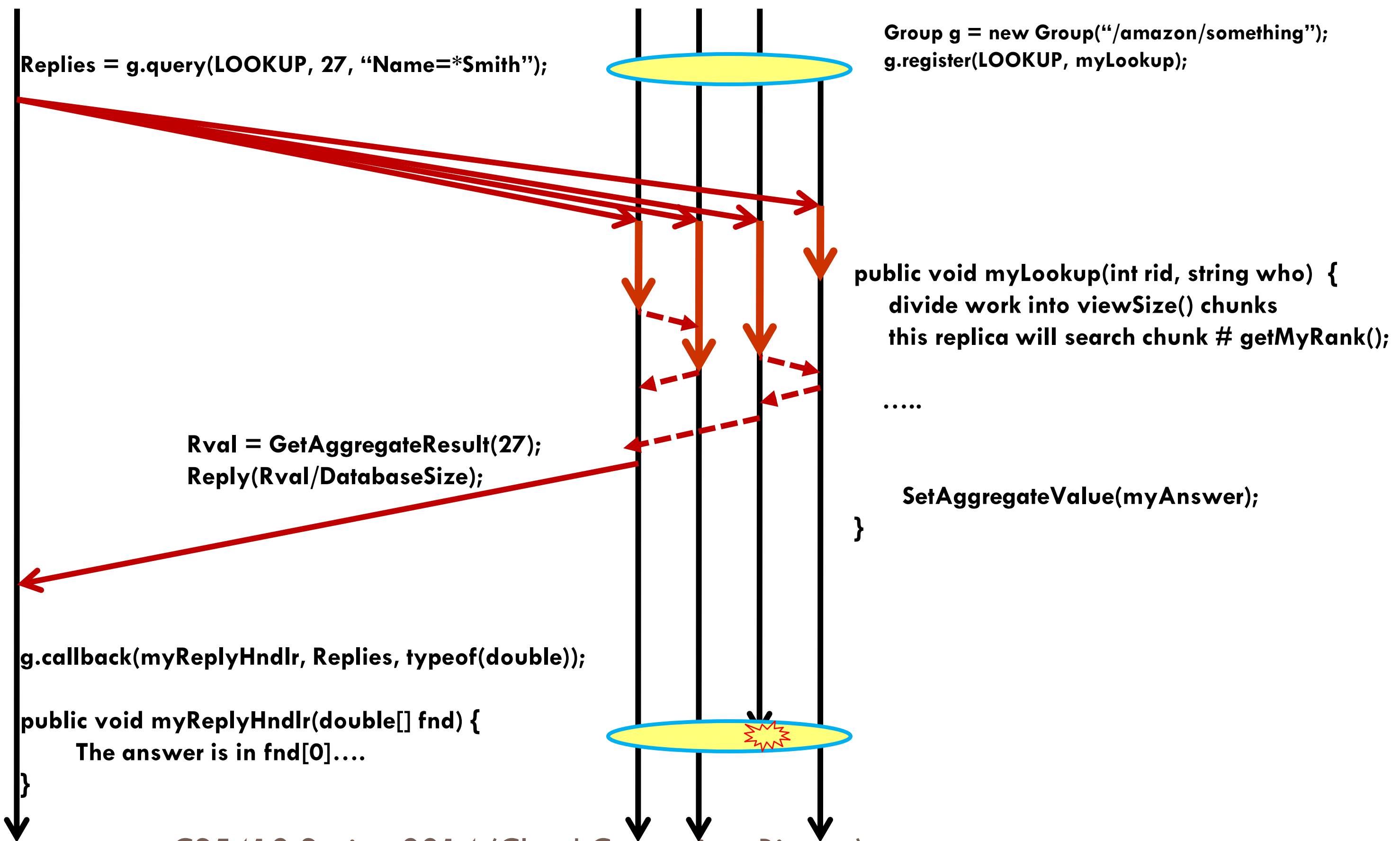
Scalable Aggregation

- Used if group is *really big*
- Request, updates: still via multicast
- *Response* is aggregated within a tree



Example: nodes {a,b,c,d} collaborate to perform a query

Aggregated Parallel search



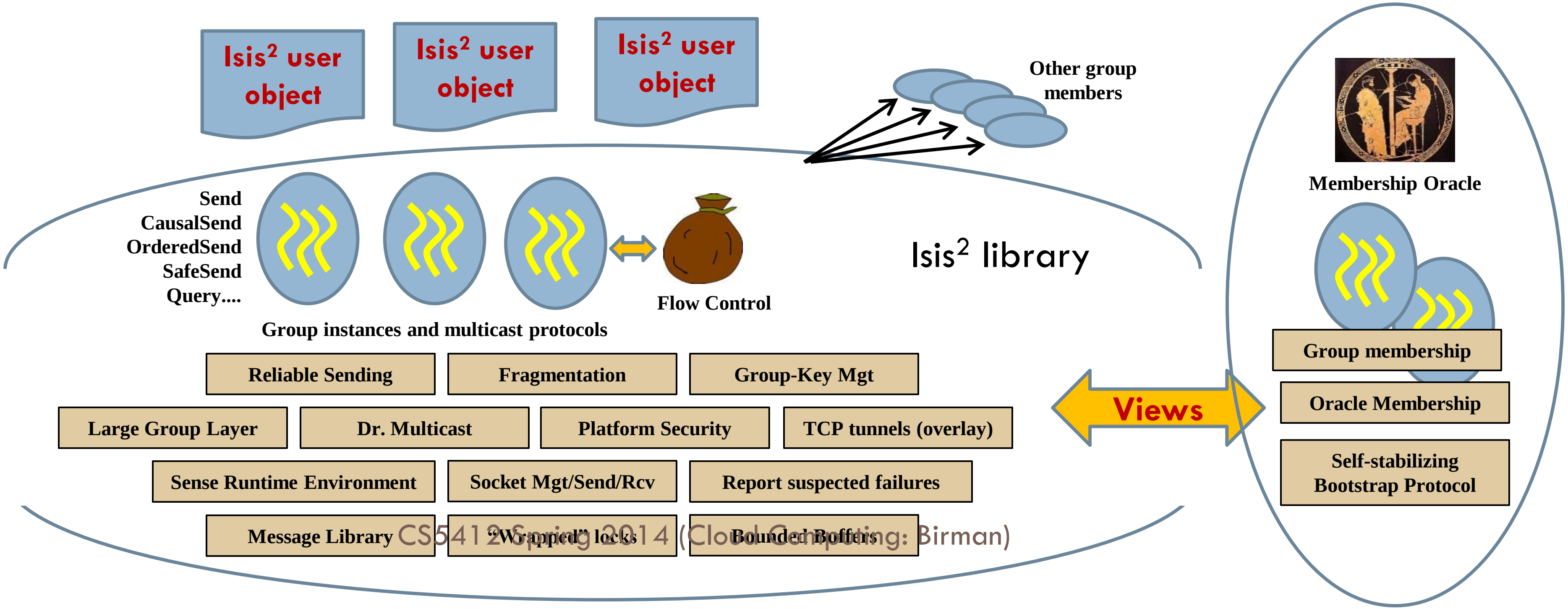
Large groups

- They can only be used in a few ways
  - All sending is actually done by the rank-0 member.
    - If others send, a relaying mechanism forwards the message via the rank-0 member
  - This use of Send does guarantee causal order: in fact it provides a causal, total ordering
  - No support for SafeSend

- Thus most of the fancy features of Isis^2 are only for use in small groups

Recall our "community" slide?

- We've seen how many (not all) of this was built!
- The system is very powerful with a wide variety of possible use styles and cases



Isis^2 offers (too) many choices!

Primitive	FIFO/Total?	Causal?	Weak/Strong Durability	Small/Large
RawSend, RawP2PSend, RawQuery	FIFO	No	Not even reliable	Either
Send, etc (same set of variants)	FIFO if underlying group is small. Total order if large.	No	Reliable, weak durability (calling Flush assures strong durability)	Either
CausalSend	FIFO+Causal	Yes	Reliable, weak	Only small
OrderedSend	Total	No	Reliable, weak	Only small
SafeSend	Total	No	Reliable, strong	Only small
Aggregated Query	Total	No	Reliable, weak	Only large

- Also: Secure/insecure, logged/not logged
- For SafeSend: # of acceptors, Disk vs. "in-memory" durability

Choice or simplicity

- Many developers just use Paxos
  - Has the strongest properties, hence a good one-size-fits-all option. SafeSend with disk durability in $Isis^2$
  - But Paxos can be slow and this is one reason CAP is applied in the first tier of the cloud

- $Isis^2$ has a wide range of options
  - Intended to permit experiments, innovative ideas
  - Pay for what you need and use… SafeSend if you like
  - … flexibility permits higher performance

Recommendation?

- We urge people who use Isis[2] to initially start with very simple applications and styles of use
  - ***Use OrderedSend for everything.***
    - Until you are familiar with the system, don't try to put a group in front of a replicated database that needs to tolerate total failure/restarts
    - Fancy features are for fancy use cases that really need them… many applications won't!