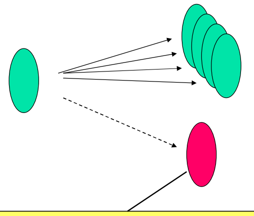
CS514: Intermediate Course in Operating Systems

Professor Ken Birman
Vivek Vishnumurthy: TA

Scalability

- Today we'll focus on how things scale
  - Basically: look at a property that matters
  - Make something "bigger"
    - Like the network, the number of groups, the number of members, the data rate
  - Then measure the property and see impact
- Often we can "hope" that no slowdown would occur. But what really happens?

Stock Exchange Problem: Sometimes, someone is slow…

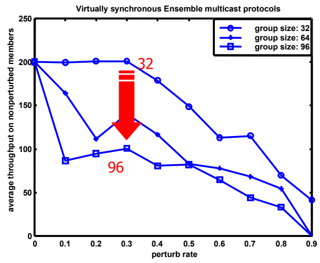


Most members are healthy….

… but one is slow

i.e. something is contending with the red process, delaying its handling of incoming messages…

With a slow receiver, throughput collapses as the system scales up



Why does this happen?

- Superficially, because data for the slow process piles up in the sender's buffer, causing flow control to kick in (prematurely)
- But why does the problem grow worse as a function of group size, with just one "red" process?
  - Small perturbations happen all the time

Broad picture?

- Virtual synchrony works well under bursty loads
- And it scales to fairly large systems (SWX uses a hierarchy to reach ~500 users)
  - From what we've seen so far, this is about as good as it gets for reliability
  - Recall that stronger reliability models like Paxos are costly and scale far worse
- Desired: steady throughput under heavy load and stress

Protocols famous for scalability

- Scalable reliable multicast (SRM)
- Reliable Multicast Transport Protocol (RMTP)
- On-Tree Efficient Recovery using Subcasting (OTERS)
- Several others: TMP, MFTP, MFTP/EC...

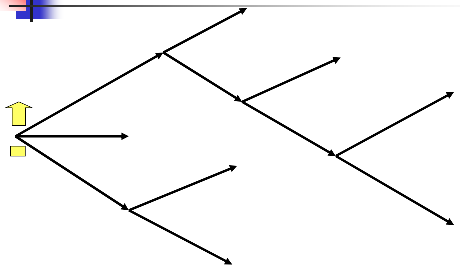
- But when stability is tested under stress, every one of these protocols collapses *just like virtual synchrony!*

Example: Scalable Reliable Multicast (SRM)

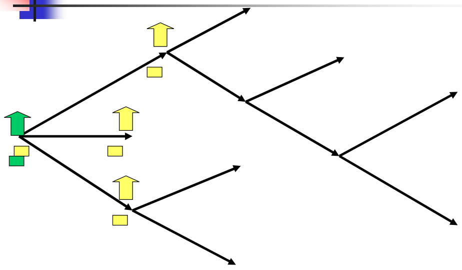
- Originated in work on Wb and Mbone
- Idea is to do "local repair" if messages are lost, various optimizations keep load low and repair costs localized
- Wildly popular for internet "push," seen as solution for Internet radio and TV
- But receiver-driven reliability model lacks "strong" reliability guarantees

---

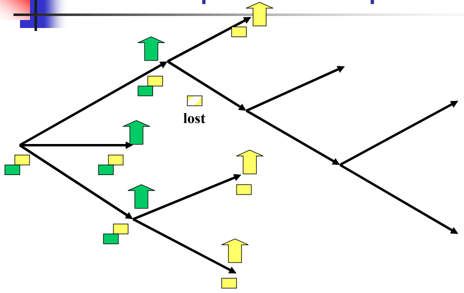
Local Repair Concept



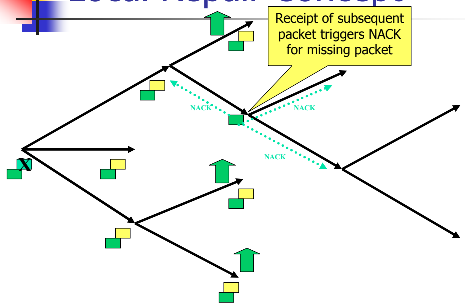
Local Repair Concept



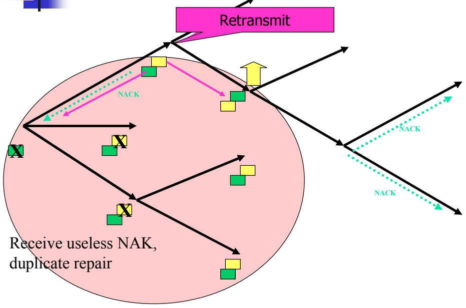
Local Repair Concept



Local Repair Concept


Receipt of subsequent packet triggers NACK for missing packet

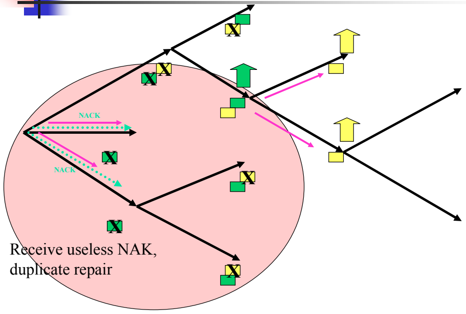
Local Repair Concept



Retransmit

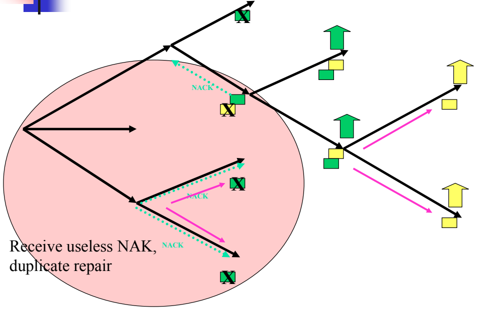
Receive useless NAK,
duplicate repair

Local Repair Concept



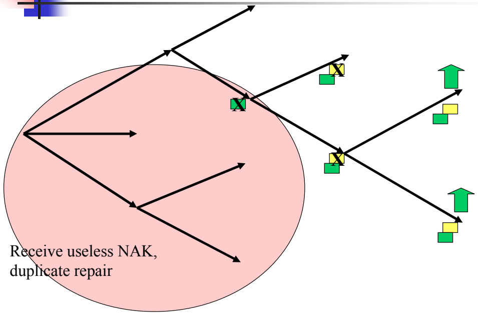
Receive useless NAK,
duplicate repair

Local Repair Concept



Receive useless NAK,
duplicate repair

Local Repair Concept



Receive useless NAK,
duplicate repair

Limitations?

- SRM runs in application, not router, hence IP multicast of nack's and retransmissions tend to reach many or all processes
- Lacking knowledge of who should receive each message, SRM has no simple way to know when a message can be garbage collected at the application layer
- Probabilistic rules to suppress duplicates

In practice?

- As the system grows large the "probabilistic suppression" fails
  - More and more NAKs are sent in duplicate
  - And more and more duplicate data message are sent as multiple receivers respond to the same NAK
- Why does this happen?

Visualizing how SRM collapses

- Think of sender as the hub of a wheel
  - Messages depart in all directions
  - Loss can occur at many places “out there” and they could be far apart…
    - Hence NAK suppression won’t work
    - Causing multiple NAKS
  - And the same reasoning explains why any one NAK is likely to trigger multiple retransmissions!
- Experiments have confirmed that SRM overheads soar with deployment size
  - Every message triggers many NAKs and many retransmissions until the network finally melts down

Dilemma confronting developers

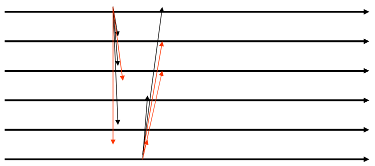
- Application is extremely critical: stock market, air traffic control, medical system
- Hence need a strong model, guarantees
- But these applications often have a soft-realtime subsystem
  - Steady data generation
  - May need to deliver over a large scale

Today introduce a new design pt.

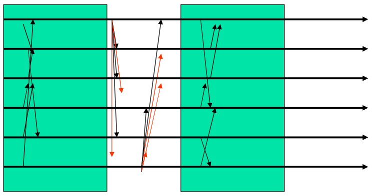
- Bimodal multicast (pbcast) is reliable in a sense that can be formalized, at least for some networks
  - Generalization for larger class of networks should be possible but maybe not easy
- Protocol is also very stable under steady load even if 25% of processes are perturbed
- Scalable in much the same way as SRM

Environment

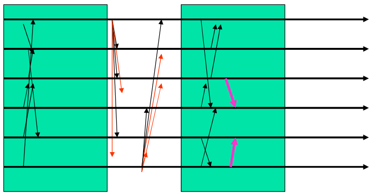
- Will assume that *most* links have known throughput and loss properties
- Also assume that *most* processes are responsive to messages in bounded time
- But can tolerate *some* flakey links and *some* crashed or slow processes.



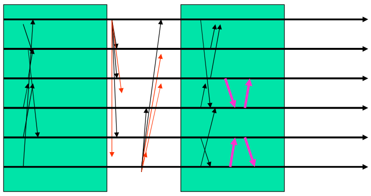
Start by using *unreliable* multicast to rapidly distribute the message. But some messages may not get through, and some processes may be faulty. So initial state involves partial distribution of multicast(s)



Periodically (e.g. every 100ms) each process sends a *digest* describing its state to some randomly selected group member. The digest identifies messages. It doesn’t include them.



Recipient checks the gossip digest against its own history and *solicits* a copy of any missing message from the process that sent the gossip



Processes respond to solicitations received during a round of gossip by retransmitting the requested message. The round lasts much longer than a typical RPC time.

Delivery? Garbage Collection?

- Deliver a message when it is in FIFO order
- Garbage collect a message when you believe that no "healthy" process could still need a copy (we used to wait 10 rounds, but now are using gossip to detect this condition)
- Match parameters to intended environment

Need to bound costs

- Worries:
  - Someone could fall behind and never catch up, endlessly loading everyone else
  - What if some process has lots of stuff others want and they bombard him with requests?
  - What about scalability in buffering and in list of members of the system, or costs of updating that list?

Optimizations

- Request retransmissions most recent multicast first
- Idea is to "catch up quickly" leaving at most one gap in the retrieved sequence

Optimizations

- Participants bound the amount of data they will retransmit during any given round of gossip. If too much is solicited they ignore the excess requests

Optimizations

- Label each gossip message with senders gossip round number
- Ignore solicitations that have expired round number, reasoning that they arrived very late hence are probably no longer correct

Optimizations

- Don't retransmit same message twice in a row to any given destination (the copy may still be in transit hence request may be redundant)

Optimizations

- Use IP multicast when retransmitting a message if several processes lack a copy
  - For example, if solicited twice
  - Also, if a retransmission is received from "far away"
  - Tradeoff: excess messages versus low latency
- Use regional TTL to restrict multicast scope

Scalability

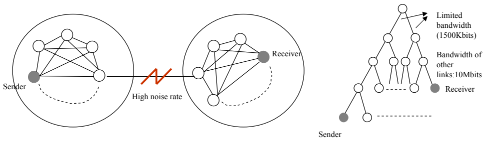
- Protocol is scalable except for its use of the membership of the full process group
- Updates could be costly
- Size of list could be costly
- In large groups, would also prefer not to gossip over long high-latency links

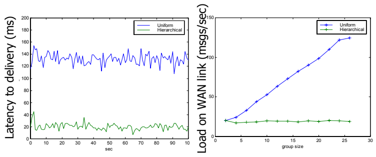
Can extend pbcast to solve both

- Could use IP multicast to send initial message. (Right now, we have a tree-structured alternative, but to use it, need to know the membership)
- Tell each process only about some subset k of the processes, k << N
- Keeps costs constant.

Router overload problem

- Random gossip can overload a central router
- Yet information flowing through this router is of diminishing quality as rate of gossip rises
- Insight: constant rate of gossip is achievable and adequate

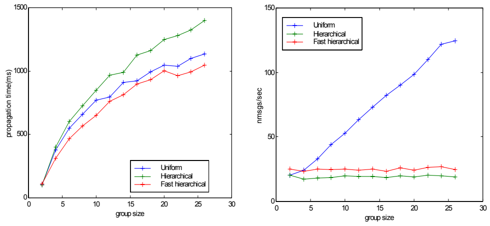




Hierarchical Gossip

- Weight gossip so that probability of gossip to a remote cluster is smaller
- Can adjust weight to have constant load on router
- Now propagation delays rise… but just increase *rate* of gossip to compensate

---



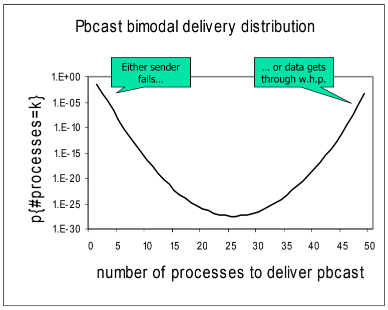
Remainder of talk

- Show results of formal analysis
  - We developed a model (won't do the math here -- nothing very fancy)
  - Used model to solve for expected reliability
- Then show more experimental data
- Real question: what would pbcast "do" in the Internet? Our experience: it works!

Idea behind analysis

- Can use the mathematics of epidemic theory to predict reliability of the protocol
- Assume an initial state
- Now look at result of running B rounds of gossip: converges exponentially quickly towards atomic delivery

---



Failure analysis

- Suppose someone tells me what they hope to "avoid"
- Model as a predicate on final system state
- Can compute the probability that pbcast would terminate in that state, again from the model

---

Two predicates

- Predicate I: A faulty outcome is one where more than 10% but less than 90% of the processes get the multicast

… Think of a probabilistic Byzantine General's problem: a disaster if many but not most troops attack

---

Two predicates

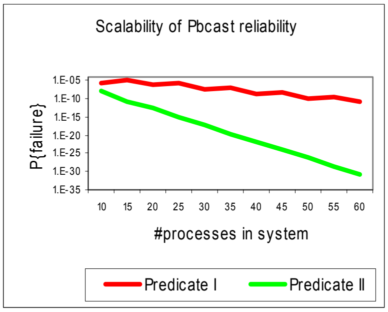
- Predicate II: A faulty outcome is one where roughly half get the multicast and failures might "conceal" true outcome

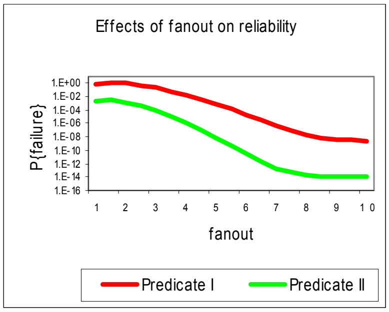
… this would make sense if using pbcast to distribute quorum-style updates to replicated data. The costly hence undesired outcome is the one where we need to rollback because outcome is "uncertain"

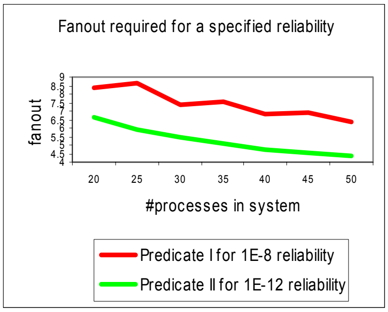
Two predicates

- Predicate I: More than 10% but less than 90% of the processes get the multicast
- Predicate II: Roughly half get the multicast but crash failures might "conceal" outcome
- Easy to add your own predicate. Our methodology supports any predicate over final system state

---







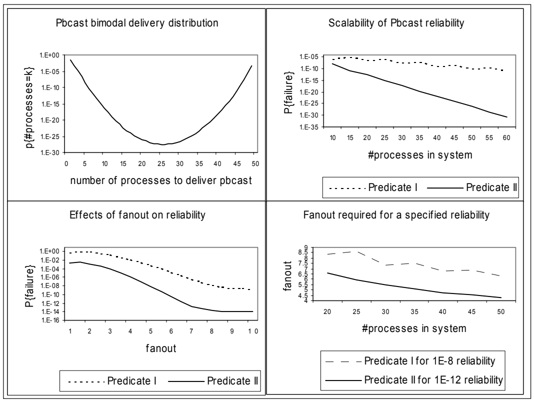


Figure 5: Graphs of analytical results

Discussion

- We see that pbcast is indeed bimodal even in worst case, when initial multicast fails
- Can easily tune parameters to obtain desired guarantees of reliability
- Protocol is suitable for use in applications where bounded risk of undesired outcome is sufficient

Model makes assumptions...

- These are rather simplistic
- Yet the model seems to predict behavior in real networks, anyhow
- In effect, the protocol is not merely robust to process perturbation and message loss, but also to perturbation of the model itself
- Speculate that this is due to the incredible power of exponential convergence...

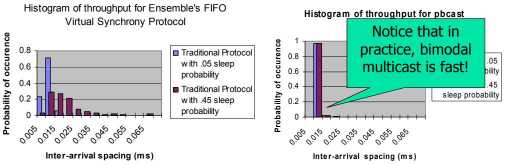
Experimental work

- SP2 is a large network
  - Nodes are basically UNIX workstations
  - Interconnect is basically an ATM network
  - Software is standard Internet stack (TCP, UDP)
- We obtained access to as many as 128 nodes on Cornell SP2 in Theory Center
- Ran pbcast on this, and also ran a second implementation on a real network

Example of a question

- Create a group of 8 members
- Perturb one member in style of Figure 1
- Now look at “stability” of throughput
  - Measure rate of received messages during periods of 100ms each
  - Plot histogram over life of experiment

Source to dest latency distributions



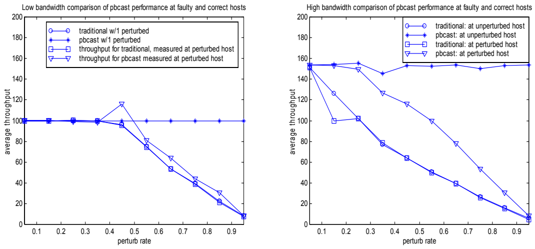
Notice that in practice, bimodal multicast is fast!

Now revisit Figure 1 in detail

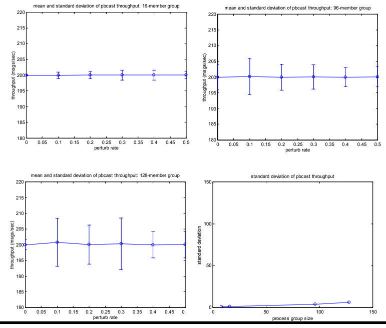
- Take 8 machines
- Perturb 1
- Pump data in at varying rates, look at rate of received messages

---

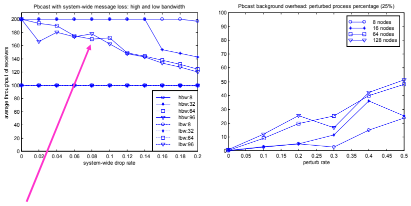
Revisit our original scenario with perturbations (32 processes)



Throughput variation as a function of scale



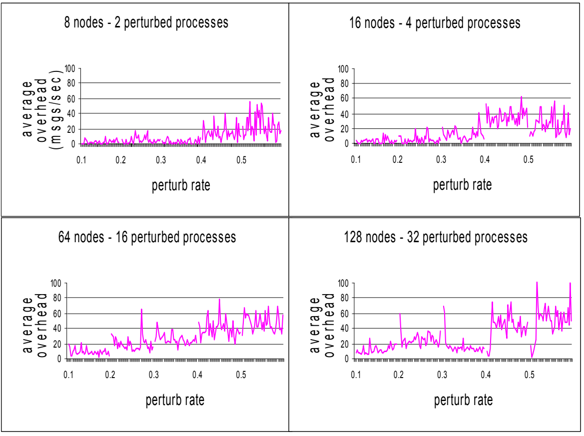
Impact of packet loss on reliability and retransmission rate



Notice that when network becomes overloaded, healthy processes experience packet loss!

What about growth of overhead?

- Look at messages other than original data distribution multicast
- Measure worst case scenario: costs at main generator of multicasts
- Side remark: all of these graphs look identical with multiple senders or if overhead is measured elsewhere….



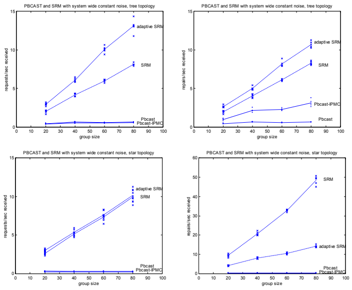
Growth of Overhead?

- Clearly, overhead does grow
- We know it will be bounded except for probabilistic phenomena
- At peak, load is still fairly low

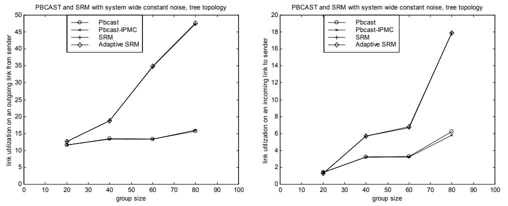
Pbcast versus SRM, 0.1% packet loss rate on all links

Tree networks

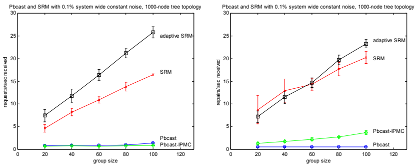
Star networks



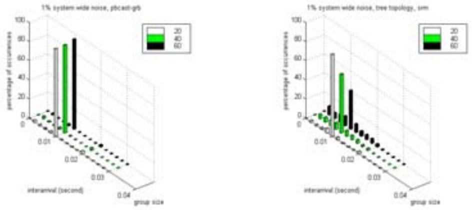
Pbcast versus SRM: link utilization



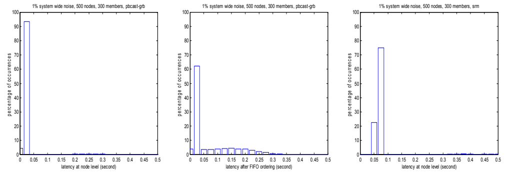
Pbcast versus SRM: 300 members on a 1000-node tree, 0.1% packet loss rate



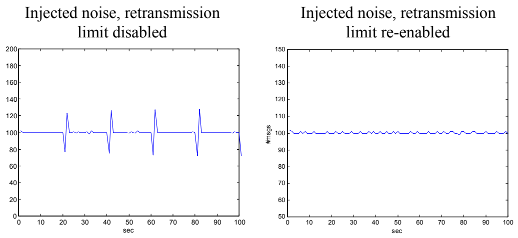
Pbcast Versus SRM: Interarrival Spacing



Pbcast versus SRM: Interarrival spacing (500 nodes, 300 members, 1.0% packet loss)

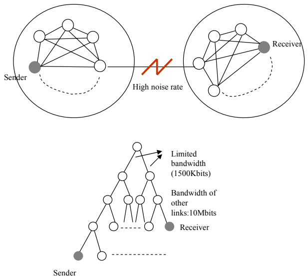


Real Data: Spinglass on a 10Mbit ethernet (35 Ultrasparc's)

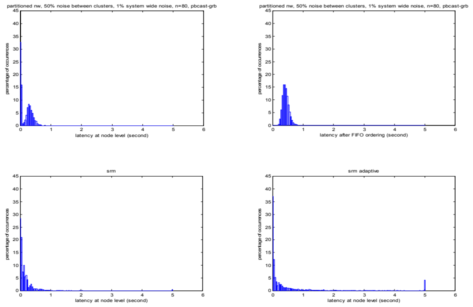

Injected noise, retransmission limit disabled

Injected noise, retransmission limit re-enabled

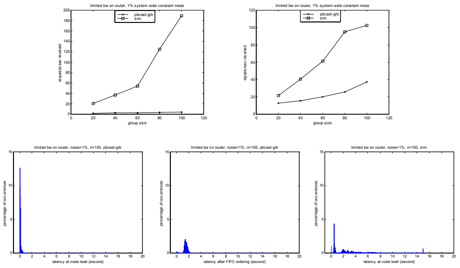
Networks structured as clusters



Delivery latency in a 2-cluster LAN,
50% noise between clusters, 1% elsewhere



Requests/repairs and latencies with bounded router bandwidth



Discussion

- Saw that stability of protocol is exceptional even under heavy perturbation
- Overhead is low and stays low with system size, bounded even for heavy perturbation
- Throughput is extremely steady
- In contrast, virtual synchrony and SRM both are fragile under this sort of attack

Programming with pbcast?

- Most often would want to *split* application into multiple subsystems
  - Use pbcast for subsystems that generate regular flow of data and can tolerate infrequent loss if risk is bounded
  - Use stronger properties for subsystems with less load and that need high availability and consistency at all times

Programming with pbcast?

- In stock exchange, use pbcast for pricing but abcast for "control" operations
- In hospital use pbcast for telemetry data but use abcast when changing medication
- In air traffic system use pbcast for routine radar track updates but abcast when pilot registers a flight plan change

Our vision: One protocol side-by-side with the other

- Use virtual synchrony for replicated data and control actions, where strong guarantees are needed for safety
- Use pbcast for high data rates, steady flows of information, where longer term properties are critical but individual multicast is of less critical importance

Summary

- New data point in a familiar spectrum
  - Virtual synchrony
  - Bimodal probabilistic multicast
  - Scalable reliable multicast
- Demonstrated that pbcast is suitable for analytic work
- Saw that it has exceptional stability