CS514: Intermediate Course in Operating Systems

Professor Ken Birman
Vivek Vishnumurthy: TA

Autonomic computing

- Hot new area intended as a response to some of our concerns
- Basic idea is to emulate behaviors common in biological systems
  - For example: if you rush to class, your heart beats faster and you might sweat a little… but this isn't something you "intend"
  - The response is an "autonomic" one

Goals for autonomic systems

- The so-called "self-*" properties
  - Self-installing
  - Self-configuring
  - Self-monitoring
  - Self-diagnosing and self-repairing
  - Adaptive when loads or resources change
- Can we create autonomic systems?

What makes it hard?

- From the inside of a system, options are often very limited
  - Last time we saw that even detecting a failure can be very hard… and that if we can't be sure, making "decisions" can be even harder!
  - Also, most systems are designed for simplicity and speed. Self-* mechanisms add complexity and overhead

Modern approach

- Perhaps better to think of external platform tools that help a system out
- The platform can automate tasks that a user might find hard to do on their own
  - Such as restart after a failure
  - You tell the platform "please keep 5 copies of my service running." If a copy crashes, it finds an unloaded node and restarts it

Slight shift in perspective

- Instead of each application needing to address these hard problems… we can shift the role to standardized software
- It may have ways to solve hard problems that end-users can't access
  - Like ways to ask hardware for "help"
  - Or lots of ways to sense status

Werner Vogels (Amazon CTO)

- Discussed "world wide failure detectors"
  - Issue: How to sense failure
  - We saw that this is hard to get right
- A neighbor's mailbox is overflowing... should you call 911?
  - Leaving the mail out isn't "proof of death"
  - Many other ways to sense "health"

How can a platform check health?

- Application exits but O/S is still running
- O/S reboots itself
- NIC card loses carrier, then regains it after communication links have broken
- O/S may have multiple communication paths.... even if application gets "locked" onto just one path
- ... the list goes on and on

Vogels proposal?

- He urges that we build large-scale "system membership services"
  - The service would track membership in the data center or network
  - Anyone talking to a component would also register themselves, and their "interests" with the service
  - It reports problems, consistently and quickly

Such a service can overcome Jack and Jill's problem!

- They couldn't agree on status
- But a service can make a rule
  - <u>*Even if an application is running, if it loses connection to a majority of the servers running the "health service", we consider it to have crashed.*</u>
- With this rule, the health abstraction can be *implemented* by the platform!

Jack and Jill with a Failure Detector

- Jack and Jill agree to check their mail at least once every ten minutes
- The failure detector, running as a system service, monitors their actions
- A failure to check mail triggers a system-wide notification
  - *Terrible news. Sad tiddings. Jack is dead!*
  - If it makes a mistake... tough luck!

How to make TCP use this

- Take TCP
  - Disable the "SO_KEEPALIVE" feature
  - Now TCP won't sense timeouts and hence will never break a connection
- Now write a wrapper
  - User makes a TCP connection... wrapper registers with the health service
  - Health problem? Break the connection...m

A health service is an autonomic construct

- How else could we build autonomic platform tools?
  - For example, could we build a tool to robustly notify all the applications when something important happens?
    - *E.g. "System overload! Please scale back all non-vital functionality"*
  - Could we build a tool to "make a map" showing the status of a large system?

Gossip: A valuable tool...

- So-called *gossip protocols* can be robust even when everything else is malfunctioning
- Idea is to build a distributed protocol a bit like gossip among humans
  - "Did you hear that Sally and John are going out?"
  - Gossip spreads like lightening...

Gossip: basic idea

- Node A encounters "randomly selected" node B (might not be *totally* random)
  - Gossip push ("rumor mongering"):
    - A tells B something doesn't know
  - Gossip pull ("anti-entropy")
    - A asks B for something it is trying to "find"
  - Push-pull gossip
    - Combines both mechanisms

Definition: A gossip protocol...

- Uses random pairwise state merge
- Runs at a steady rate (and this rate is much slower than the network RTT)
- Uses bounded-size messages
- Does not depend on messages getting through reliably

Gossip benefits... and limitations

- Information flows around disruptions
- Scales very well
- Typically reacts to new events in log(N)
- Can be made self-repairing

- Rather slow
- Very redundant
- Guarantees are at best probabilistic
- Depends heavily on the randomness of the peer selection

For example

- We could use gossip to track the health of system components
- We can use gossip to report when something important happens
- In the remainder of today's talk we'll focus on event notifications. Next week we'll look at some of these other uses

Typical push-pull protocol

- Nodes have some form of database of participating machines
  - Could have a hacked bootstrap, then use gossip to keep this up to date!
- Set a timer and when it goes off, select a peer within the database
  - Send it some form of "state digest"
  - It responds with data you need and its own state digest
  - You respond with data it needs

Where did the "state" come from?

- The data eligible for gossip is usually kept in some sort of table accessible to the gossip protocol
- This way a separate thread can run the gossip protocol
- It does upcalls to the application when incoming information is received

Gossip often runs over UDP

- Recall that UDP is an "unreliable" datagram protocol supported in internet
  - Unlike for TCP, data can be lost
  - Also packets have a maximum size, usually 4k or 8k bytes (you can control this)
  - Larger packets are more likely to get lost!
- What if a packet would get too large?
  - Gossip layer needs to pick the most valuable stuff to include, and leave out the rest!

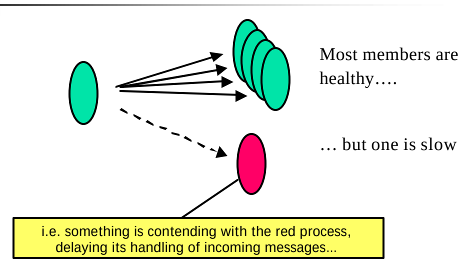
Algorithms that use gossip

- Gossip is a hot topic!
- Can be used to…
  - Notify applications about some event
  - Track the status of applications in a system
  - Organize the nodes in some way (like into a tree, or even sorted by some index)
  - Find "things" (like files)
- Let's look closely at an example

Bimodal multicast

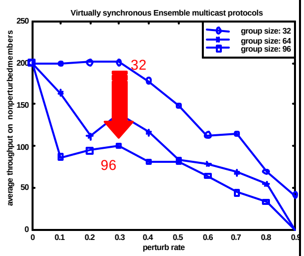
- This is Cornell work from about 10 years ago
- Goal is to combine gossip with UDP (also called IP) multicast to make a very robust multicast protocol

Stock Exchange Problem: Sometimes, someone is slow…



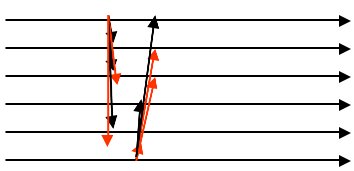
With classical reliable multicast, throughput collapses as the system scales up!

- Even if we have just one slow receiver… as the group gets larger (hence more *healthy* receivers), impact of a performance perturbation is more and more evident!

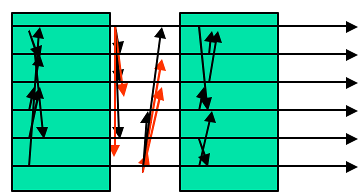


Why does this happen?

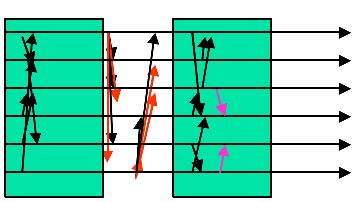
- Most reliable multicast protocols are based on an ACK/NAK scheme (like TCP but with multiple receivers). Sender retransmits lost packets.
- As number of receivers gets large ACKS/NAKS pile up (sender has more and more work to do)
  - Hence it needs longer to discover problems
  - And this causes it to buffer messages for longer and longer… hence flow control kicks in!
  - So the whole group slow down



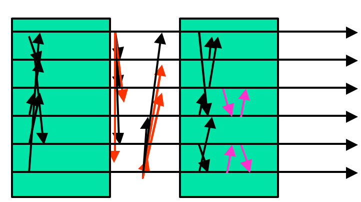
Start by using *unreliable UDP* multicast to rapidly distribute the message. But some messages may not get through, and some processes may be faulty. So initial state involves partial distribution of multicast(s)



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
  - Report an unrecoverable loss if a gap persists for so long that recovery is deemed "impractical"
- 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

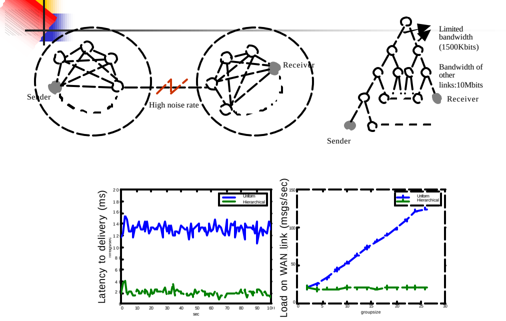
- Use UDP multicast when retransmitting a message if several processes lack a copy
  - For example, if solicited twice
  - Also, if a retransmission is received from “far away”
  - Tradeoff: excess messages versus low latency
- Use regional TTL to restrict multicast scope

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

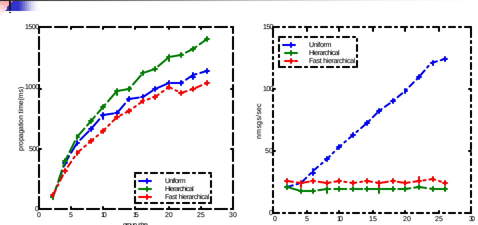
Router overload problem

- 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



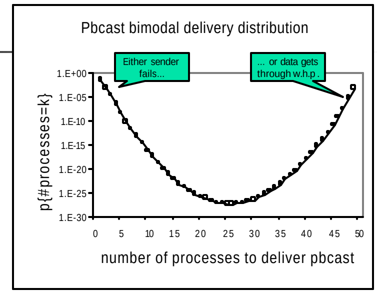
Hierarchical Gossip

- 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



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*

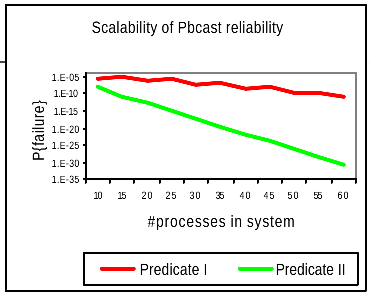
Two predicates

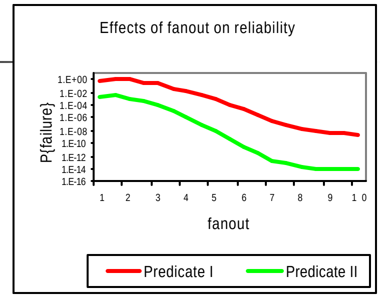
- Predicate II: A faulty outcome is one where roughly half get the multicast and failures might "conceal" true outcome

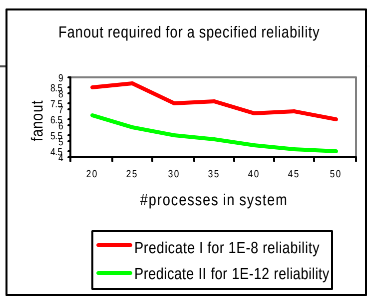
… *this would make sense if using pbcast to distribute quorum-style updates to replicated data. The costly hence undesired outcome is the one where we need to rollback because outcome is "uncertain"*

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







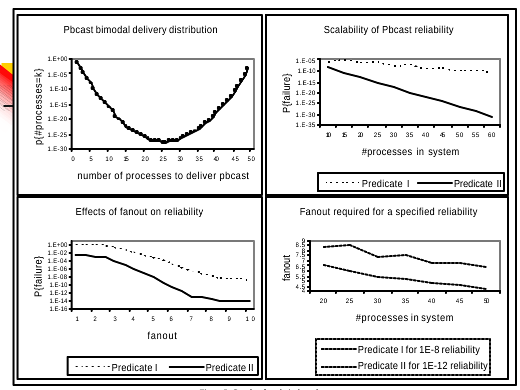


Figure 5: Graphs of analytical results

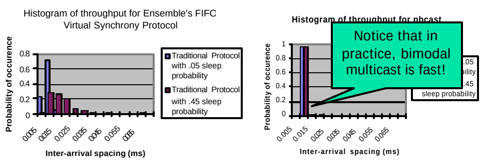
Experimental work

- 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

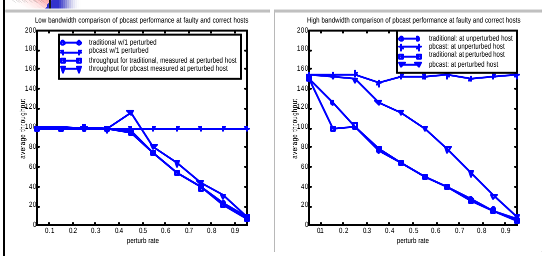
Source to dest latency distributions



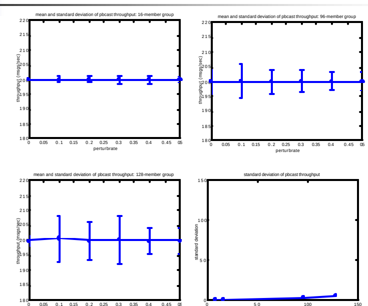
Now revisit Figure 1 in detail

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

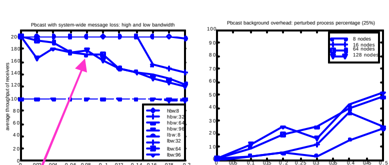
Revisit our original scenario with perturbations (32 processes)



Throughput variation as a function of scale



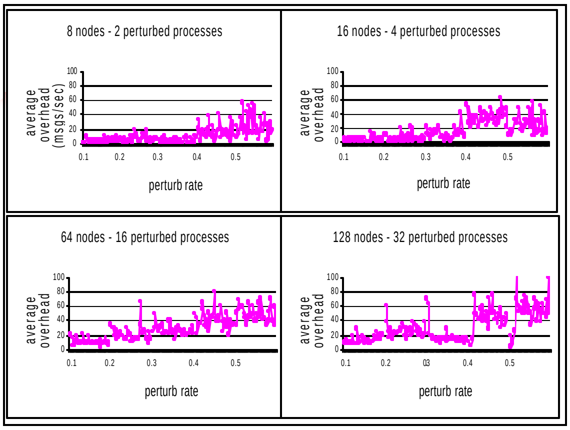
Impact of packet loss on reliability and retransmission rate



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….



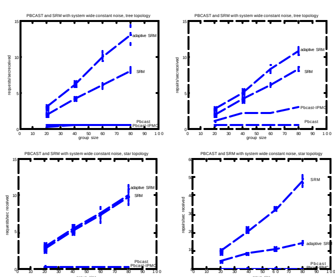
Growth of Overhead?

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

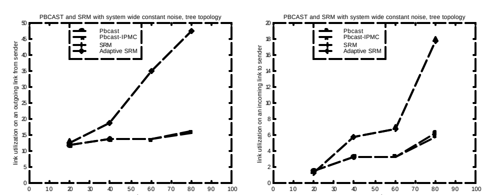
Pbcast versus SRM, 0.1% packet loss rate on all links

Tree networks

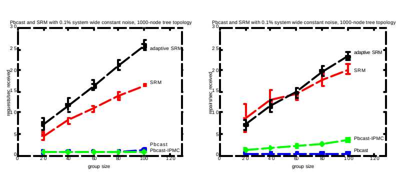
Star networks



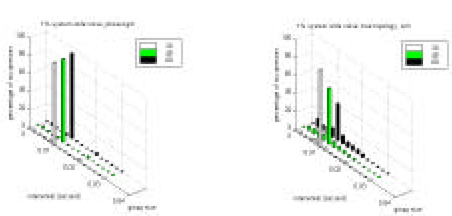
Pbcast versus SRM: link utilization



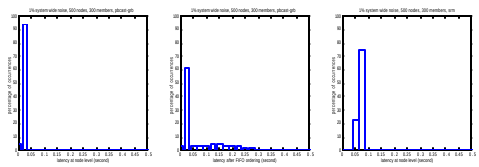
Pbcast versus SRM: 300 members on a 1000-node tree, 0.1% packet loss rate



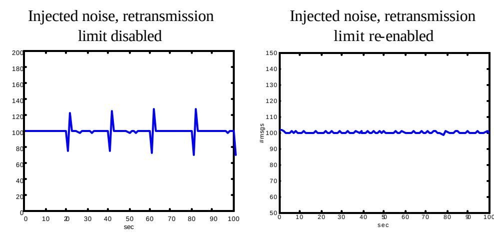
Pbcast Versus SRM: Interarrival Spacing



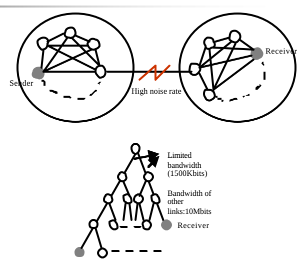
Pbcast versus SRM: Interarrival spacing (500 nodes, 300 members, 1.0% packet loss)



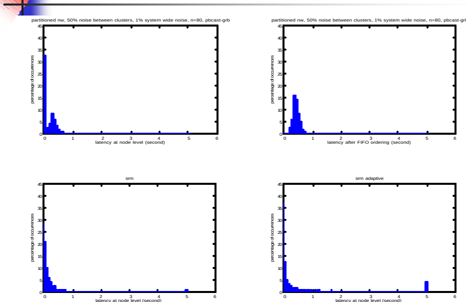
Real Data: Bimodal Multicast on a 10Mbit ethernet (35 Ultrasparc's)



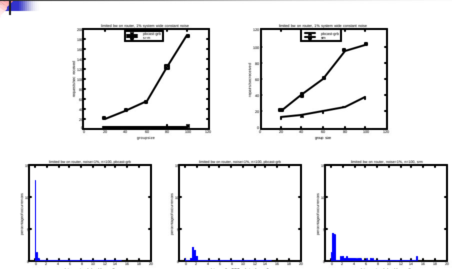
Networks structured as clusters



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



Requests/repairs and latencies with bounded router bandwidth



Summary

- Gossip is a valuable tool for addressing some of the needs of modern autonomic computing
- Often paired with other mechanisms, eg anti-entropy paired with UDP multicast
- Solutions scale well (if well designed!)