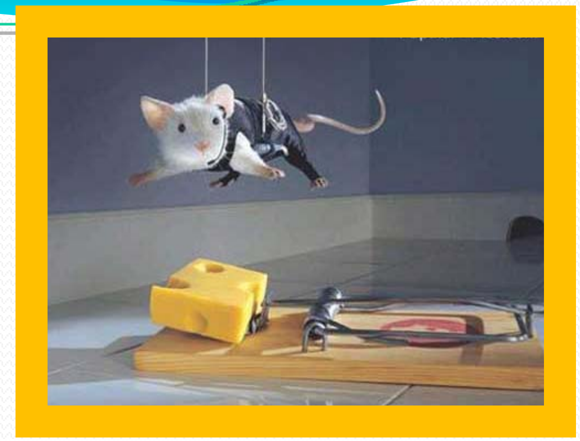
Designing a New Multicast Infrastructure for Linux

Ken Birman

Cornell University. CS5410 Fall 2008.

Mission Impossible…



- Today, multicast is *persona non-grata* in most cloud settings
  - Amazon's stories of their experience with violent load oscillations has frightened most people in the industry
  - They weren't the only ones…
- Today:
  - Design a better multicast infrastructure for using the Linux Red Hat operating system in enterprise settings
  - Target: trading floor in a big bank (if any are left) on Wall Street, cloud computing in data centers

What do they need?

- Quick, scalable, pretty reliable message delivery
  - Argues for IPMC or a protocol like Ricochet
  - Virtual synchrony, Paxos, transactions: all would be examples of higher level solutions running *over* the basic layer we want to design

- But we don’t want our base layer to misbehave

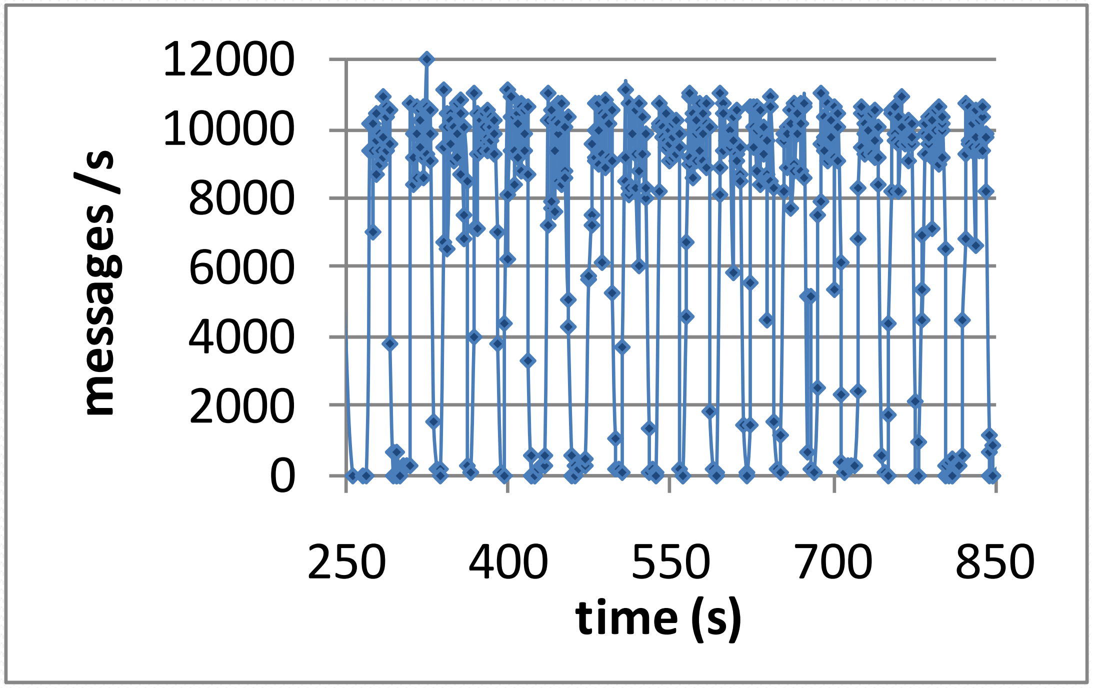
Reminder: What goes wrong?

- Earlier in the semester we touched on the issues with IPMC in existing cloud platforms
  - Applications unstable, exhibit violent load swings
  - Usually totally lossless, but sometimes drops zillions of packets all over the place
  - Various forms of resource exhaustion
- Start by trying to understand the big picture: why is this happening?

Misbehavior pattern

- Noticed when an application-layer solution, like a virtual synchrony protocol, begins to exhibit wild load swings for no obvious reason
  - For example, we saw this in QSM (Quicksilver Scalable Multicast)
  - Fixing the problem at the end-to-end layer was really hard!

QSM oscillated in this 200-node experiment when its damping and prioritization mechanisms were disabled



Tracking down the culprit

- Why was QSM acting this way?
  - When we started work, this wasn't easy to fix…
  - … issue occurred only with 200 nodes and high data rates
- But we tracked down a pattern
  - Under heavy load, the network was delivering packets to our receivers faster than they could handle them
  - Caused kernel-level queues to overflow… hence wide loss
  - Retransmission requests and resends made things worse
  - So: goodput drops to zero, overhead to infinity. Finally problem repaired and we restart… only to do it again!

Aside: QSM works well now

- We did all sorts of things to stabilize it
  - Novel "minimal memory footprint" design
  - Incredibly low CPU loads minimize delays
  - Prioritization mechanisms ensure that lost data is repaired first, before new good data piles up behind gap

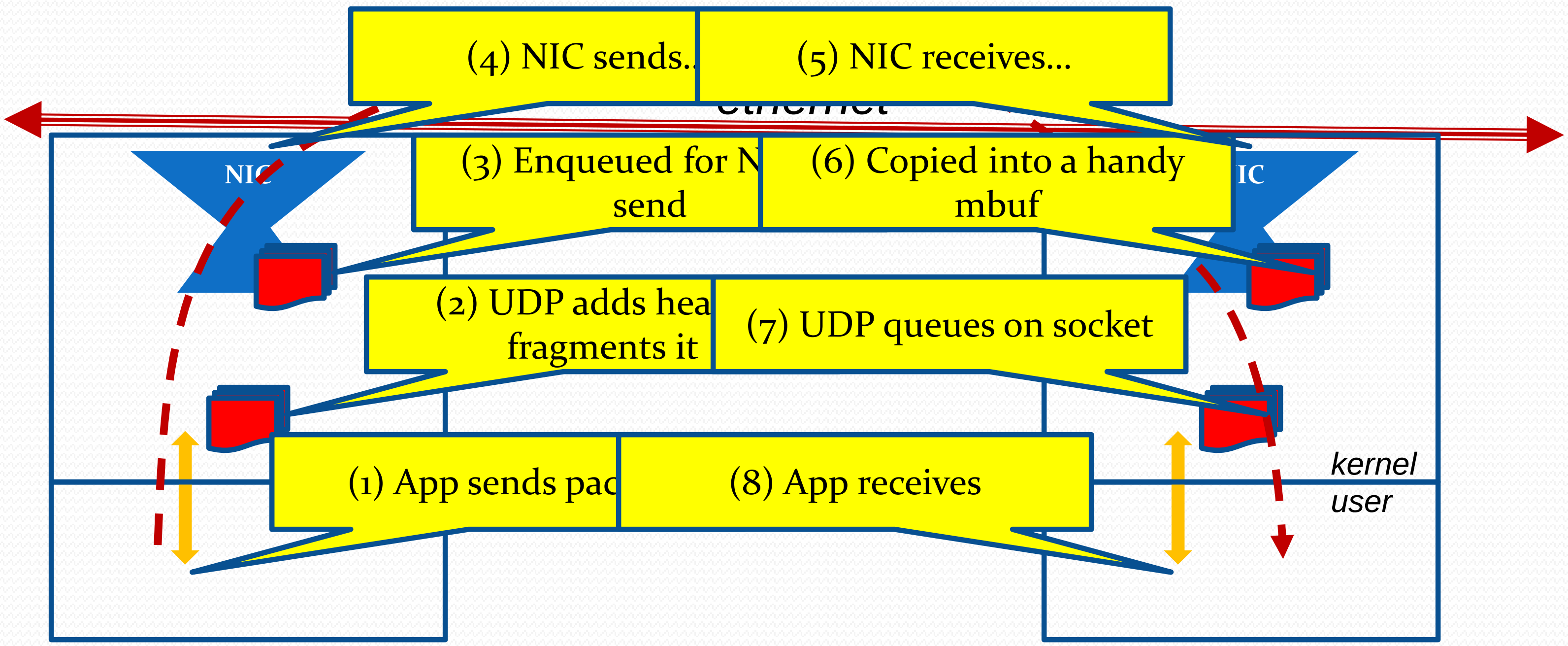
- But most systems lack these sorts of unusual solutions
  - Hence most systems simply destabilize, like QSM did before we studied and fixed these issues!
  - Linux goal: a system-wide solution

Assumption?

- Assume that if we enable IP multicast
  - Some applications will use it heavily
  - Testing will be mostly on smaller configurations
- Thus, as they scale up and encounter loss, many will be at risk of oscillatory meltdowns
  - Fixing the protocol is obviously the best solution…
  - … but we want the data center (the cloud) to also protect itself against disruptive impact of such events!

So why did receivers get so lossy?

- To understand the issue, need to understand history of network speeds and a little about the hardware



Network speeds

- When Linux was developed, Ethernet ran at 10Mbits and NIC was able to keep up
  - Then network sped up: 100Mbits common, 1Gbit more and more often seen, 10 or 40 "soon"
  - *But typical PCs didn't speed up remotely that much!*
- Why did PC speed lag?
  - Ethernets transitioned to optical hardware
  - PCs are limited by concerns about heat, expense. Trend favors multicore solutions that run slower… so why invest to create a NIC that can run faster than the bus?

NIC as a “rate matcher”

- Modern NIC has two sides running at different rates
  - Ethernet side is blazingly fast, uses ECL memory…
  - Main memory side is slower
- So how can this work?
  - Key insight: NIC usually receives one packet, but then doesn’t need to accept the “next” packet.
  - Gives it time to unload the incoming data
  - But why does it get away with this?

NIC as a “rate matcher”

- When would a machine get several back-to-back packets?
  - Server with many clients
  - Pair of machines with a stream between them: but here, limited because the sending NIC will run at the speed of its interface to the machine’s main memory – in today’s systems, usually 100MBits

- In a busy setting, only servers are likely to see back-to-back traffic, and even the server is unlikely to see a long run packets that it needs to accept!

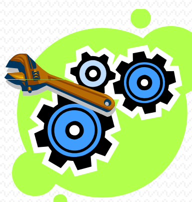
… So normally

- NIC sees big gaps between messages it needs to accept
- This gives us time…
  - …. for OS to replenish the supply of memory buffers
  - …. to hand messages off to the application

- In effect, the whole “system” is well balanced
  - But notice the hidden assumption:
  - *All of this requires that most communication be point-to-point… with high rates of multicast, it breaks down!*

Multicast: wrench in the works

- What happens when we use multicast heavily?
  - A NIC that on average received 1 out of k packets suddenly might receive many in a row (just thinking in terms of the "odds")
  - Hence will see far more back-to-back packets



- But this stresses our speed limits
  - NIC kept up with fast network traffic partly because it rarely needed to accept a packet… letting it match the fast and the slow sides…
  - With high rates of incoming traffic we overload it

Intuition: like a highway off-ramp

- With a real highway, cars just end up in a jam
- With a high speed optical net coupled to a slower NIC, packets are dropped by receiver!

More NIC worries

- Next issue relates to implementation of multicast

- Ethernet NIC actually is a pattern match machine
  - Kernel loads it with a list of {mask,value} pairs
  - Incoming packet has a destination address
  - Computes (dest&mask)==value and if so, accepts

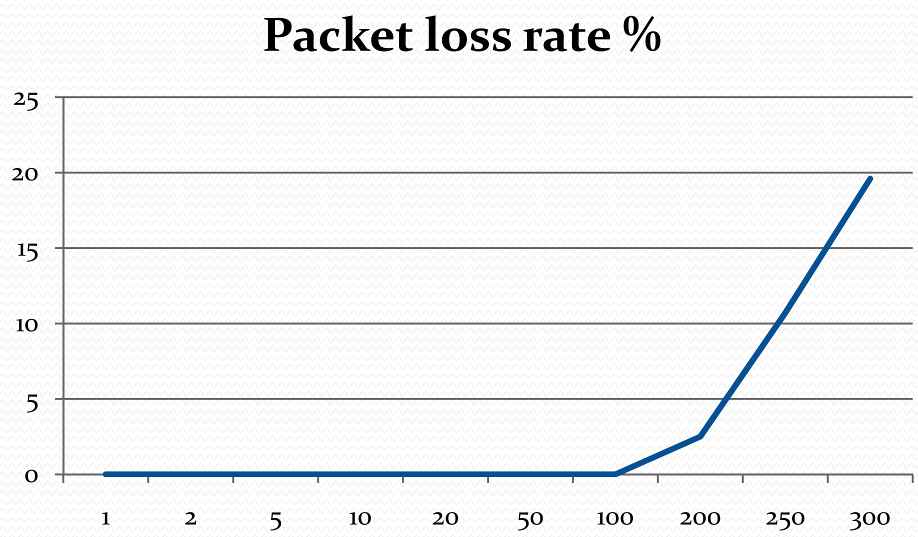
- Usually has 8 or 16 such pairs available

More NIC worries

- If the set of patterns is full… kernel puts NIC into what we call “promiscuous” mode
  - It starts to accept *all* incoming traffic
  - Then OS protocol stack makes sense of it
    - If not-for-me, ignore
  - But this requires an interrupt and work by the kernel
- All of which adds up to sharply higher
  - CPU costs (and slowdown due to cache/TLB effects)
  - Loss rate, because the more packets the NIC needs to receive, the more it will drop due to overrunning queues

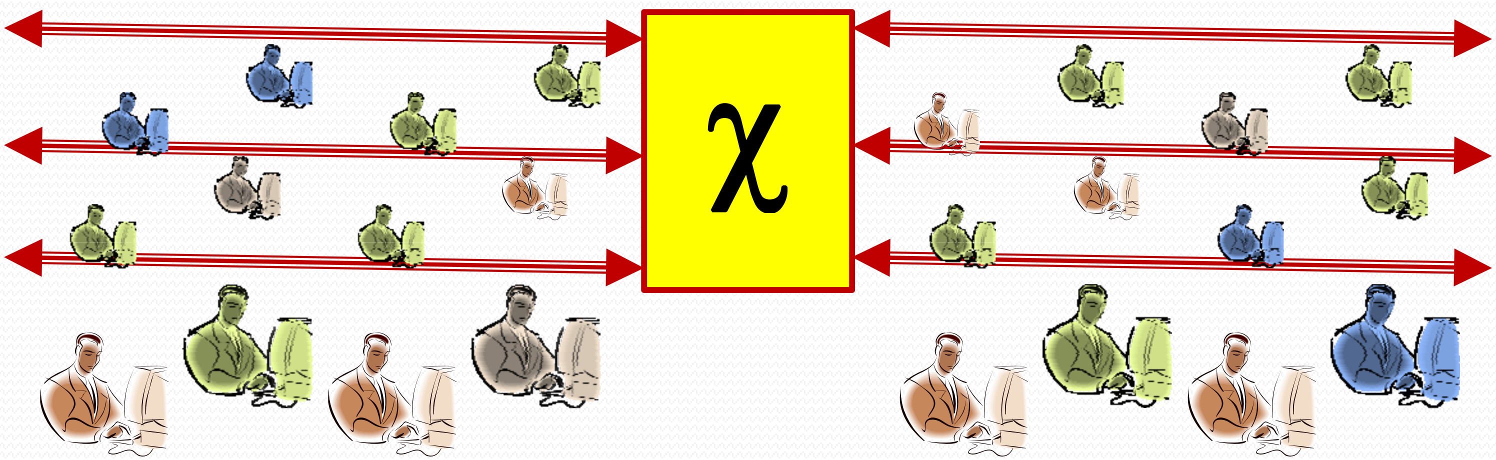
More NIC worries

- We can see this effect in an experiment done by Yoav Tock at IBM Research in Haifa



What about the switch/router?

- Modern data centers used a switched network architecture



- Question to ask: how does a switch handle multicast?

Concept of a Bloom filter

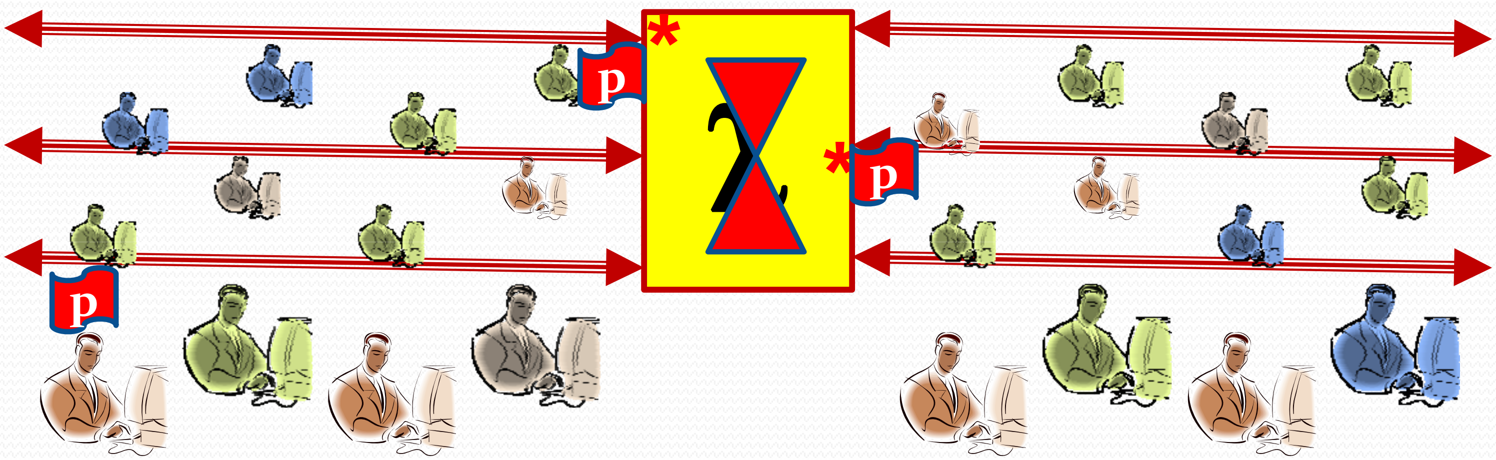
- Goal of router?
  - Packet p arrives on port a. Quickly decide which port(s) to forward it on
- Bit vector filter approach
  - Take IPMC address of p, hash it to a value in some range like [0..1023]
  - Each output port has an associated bit vector... Forward p on each port with that bit set
- Bitvector -> Bloom filter
  - Just do the hash multiple times, test against multiple vectors. Must match in all of them (reduces collisions)

Concept of a Bloom filter

- So… take our class-D multicast address (233.0.0.0/8)
  - **<u>233.</u>**17.31.129… hash it 3 times to a bit number
  - Now look at outgoing link A
    - Check bit 19 in [….010101001000000010**1**000001010100000000100000….]
    - Check bit 33 in […. **1**0100000101010000000101010010000000100000….]
    - Check bit 8 in [….0000001010100000001101010010000000**1**0100000..]
    - … all matched, so we relay a copy
  - Next look at outgoing link B
    - … match failed
  - … ETC

What about the switch/router?

- Modern data centers used a switched network architecture



- Question to ask: how does a switch handle multicast?

Aggressive use of multicast

- Bloom filters "fill up" (all bits set)
  - Not for a good reason, but because of hash conflicts
- Hence switch becomes promiscuous
  - Forwards every multicast on every network link

- Amplifies problems confronting NIC, especially if NIC itself is in promiscuous mode

Worse and worse…

- Most of these mechanisms have long memories
  - Once an IPMC address is used by a node, the NIC tends to retain memory of it, and the switch does, for a long time
  - This is an artifact of a “stateless” architecture
    - Nobody remembers *why* the IPMC address was in use
    - Application can leave but no “delete” will occur for a while
- Underlying mechanisms are lease based: periodically “replaced” with fresh data (but not instantly)

…pulling the story into focus

- We've seen that multicast loss phenomena can ultimately be traced to two major factors
  - Modern systems have a serious rate mismatch vis-à-vis the network
  - Multicast delivery pattern and routing mechanisms scale poorly
- A better Linux architecture needs to
  - Allow us to cap the rate of multicasts
  - Allow us to control which apps can use multicast
  - Control allocation of a limited set of multicast groups

Dr. Multicast (the MCMD)

- Rx for your multicast woes

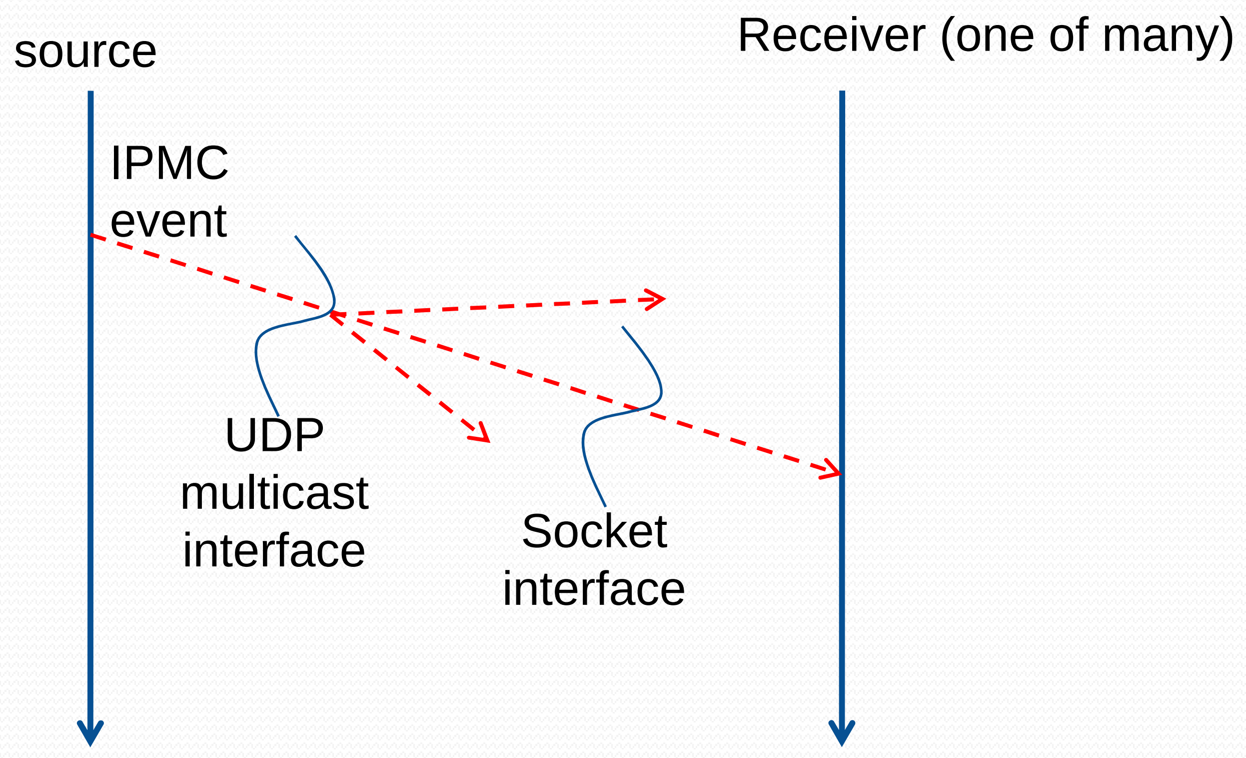
- Intercepts use of IPMC
  - Does this by *library interposition* exploiting a feature of DLL linkage
  - Then maps the logical IPMC address used by the application to either
    - A set of point-to-point UDP sends
    - A physical IPMC address, for lucky applications
  - Multiple groups share same IPMC address for efficiency

Criteria used

- Dr Multicast has an "acceptable use policy"
  - Currently expressed as low-level firewall type rules, but could easily integrate with higher level tools
- Examples
  - Application such-and-such can/cannot use IPMC
  - Limit the system as a whole to 50 IPMC addresses
- Can revoke IPMC permission rapidly in case of trouble

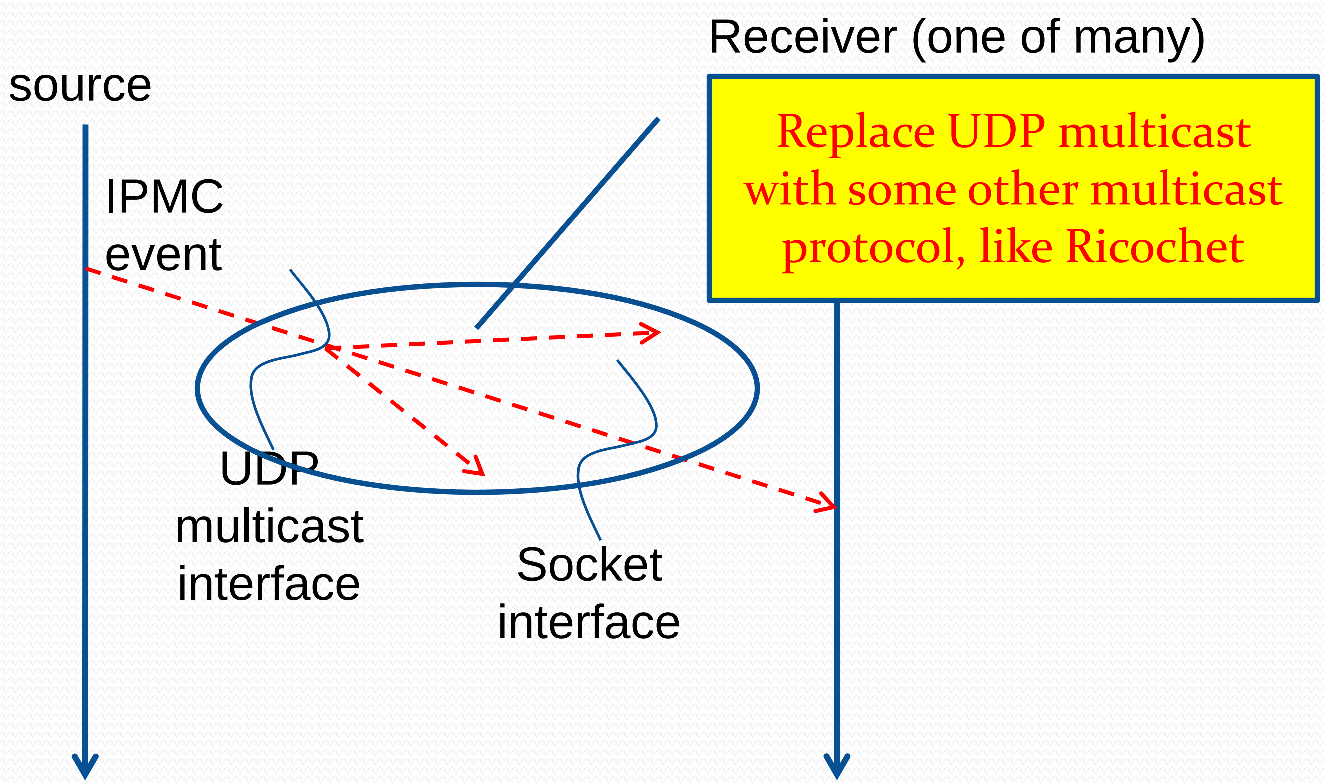
How it works

- Application uses IPMC



How it works

- Application uses IPMC



UDP multicast interface

- Very similar: With UDP
  - Socket() – creates a socket
  - Bind() connects that socket to the UDP multicast distribution network
  - Sendmsg/recvmsg() – send data

UDP multicast interface

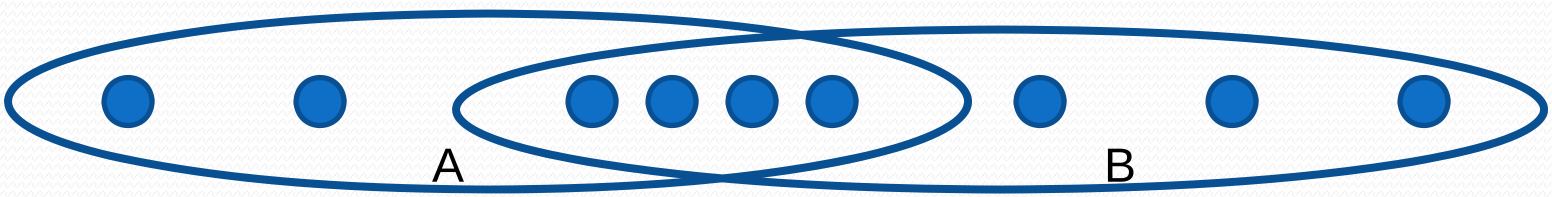
- Very similar: With UDP
  - Socket() – creates a socket
  - Bind() connects that socket to the UDP multicast distribution network
  - Sendmsg/recvmsg() – send data

Mimicry

- Many options could mimic IPMC
  - Point to point UDP or TCP, or even HTTP
  - Overlay multicast
  - Ricochet (adds reliability)
- MCMD can potentially swap any of these in under user control

Optimization

- Problem of finding an optimal group to IPMC mapping is surprisingly hard
  - Goal is to have an "exact mapping" (apps receive exactly the traffic they should receive). Identical groups get the same IPMC address
  - But can also fragment some groups....



  - Should we give an IPMC address to A, to B, to $A \cap B$?
- Turns out to be NP complete!

Greedy heuristic

- Dr Multicast currently uses a greedy heuristic
  - Looks for big, busy groups and allocates IPMC addresses to them first
  - Limited use of group fragmentation
  - We've explored more aggressive options for fragmenting big groups into smaller ones, but quality of result is very sensitive to properties of the pattern of group use

- Solution is fast, not optimal, but works well

Flow control

- How can we address rate concerns?
  - A good way to avoid broadcast storms is to somehow suppose an AUP of the type "at most xx IPMC/sec"

- Two sides of the coin
  - Most applications are greedy and try to send as fast as they can… but would work on a slower or more congested network.
    - For these, we can safely "slow down" their rate
  - But some need guaranteed real-time delivery
    - Currently can't even specify this in Linux

Flow control

- Approach taken in Dr Multicast
  - Again, starts with an AUP
    - Puts limits on the aggregate IPMC rate in the data center
    - And can exempt specific applications from rate limiting
- Next, senders in a group monitor traffic in it
  - Conceptually, happens in the network driver
- Use this to apportion limited bandwidth
  - Sliding scale: heavy users give up more

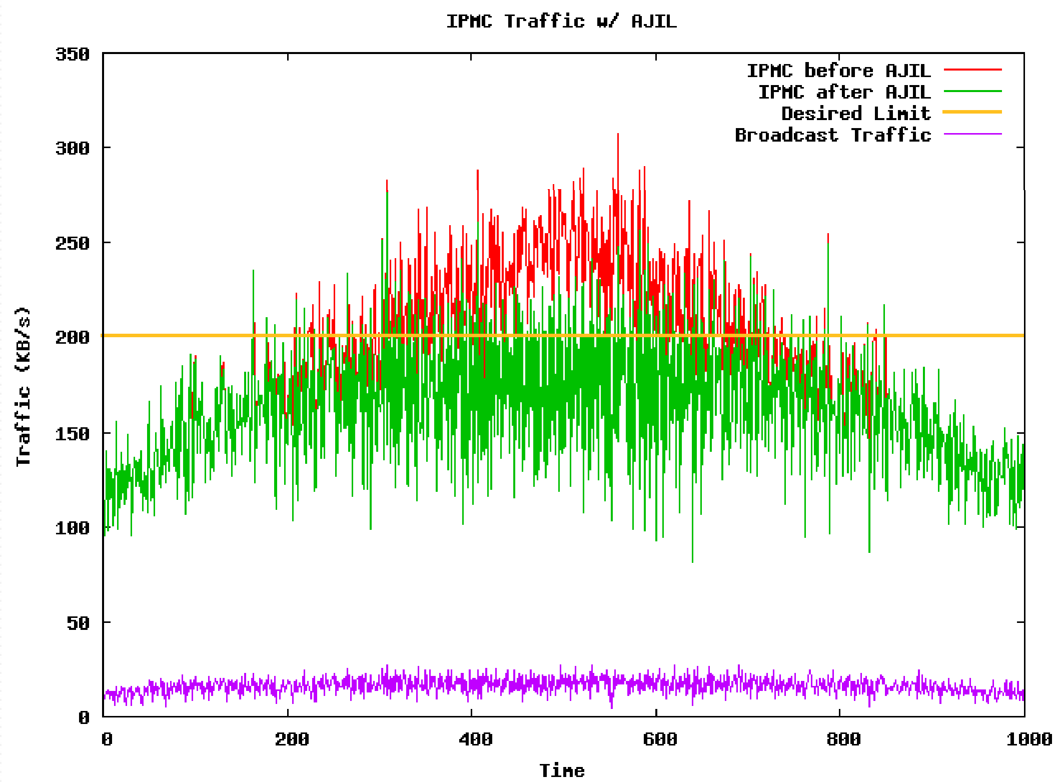
Flow control

- To make this work, the kernel send layer can delay sending packets…
  - … and to prevent application from overrunning the kernel, delay the application
  - For sender using non-blocking mode, can drop packets if sender side becomes overloaded

- Highlights a weakness of the standard Linux interface
  - No easy way to send “upcalls” notifying application when conditions change, congestion arises, etc

The "AJIL" protocol in action

- Protocol adds a rate limiting module to the Dr Multicast stack
- Uses a gossip-like mechanism to figure out the rate limits
- Work by Hussam Abu-Libdeh and others in my research group



Fast join/leave patterns

- Currently Dr Multicast doesn't do very much if applications thrash by joining and leaving groups rapidly
  - We have ideas on how to rate limit them, and it seems like it won't be hard to support
  - Real question is: how *should* this behave?

End to End philosophy / debate

- In the dark ages, E2E idea was proposed as a way to standardize rules for what should be done in the network and what should happen at the endpoints
- In the network?
  - Minimal mechanism, no reliability, just routing
  - (Idea is that anything more costs overhead yet end points would need the same mechanisms anyhow, since best guarantees will still be too weak)
- End points do security, reliability, flow control

A religion… but inconsistent…

- E2E took hold and became a kind of battle cry of the Internet community
- But they don't always stick with their own story
  - Routers drop packets when overloaded
  - TCP assumes this is the main reason for loss and backs down
- When these assumptions break down, as in wireless or WAN settings, TCP "out of the box" performs poorly

E2E and Dr Multicast

- How would the E2E philosophy view Dr Multicast?
  - On the positive side, the mechanisms being interposed operate mostly on the edges and under AUP control
  - On the negative side, they are network-wide mechanisms imposed on all users

- Original E2E paper had exceptions, perhaps this falls into that class of things?
  - *E2E except when doing something something in the network layer brings big win, costs little, and can't be done on the edges in any case...*

Summary

- Dr Multicast brings a vision of a new world of controlled IPMC
  - Operator decides who can use it, when, and how much
  - Data center no longer at risk of instability from malfunctioning applications
  - Hence operator allows IPMC in: trust (but verify, and if problems emerge, intervene)

- Could reopen door for use of IPMC in many settings