SCC.311: Group Communication

Prof. Barry Porter

Last time...

- We looked at *remote invocation* as a simple middleware layer, with several specific examples
  - This kind of distributed interaction is usually between two computers: a single client, and a single server
  - We would refer to this as a **unicast** communication style
  - We also saw that we could have a range of different levels of guarantee around whether messages are received, and received just once

Overview of communication models

- Unicast
  - One-to-one communication, the most common style for Web applications
- Broadcast
  - One-to-everyone
- Multicast
  - One-to-many
  - Many-to-many

Overview for today

- Group communication is another common middleware layer in distributed systems, with a wide range of uses
  - It also has a set of different levels of guarantee that we can offer, depending on how we implement the underlying protocol within the middleware

- We'll look at the general concept-level, then focus on:
  - *Reliable multicast*
  - *Atomic multicast*

Group communication

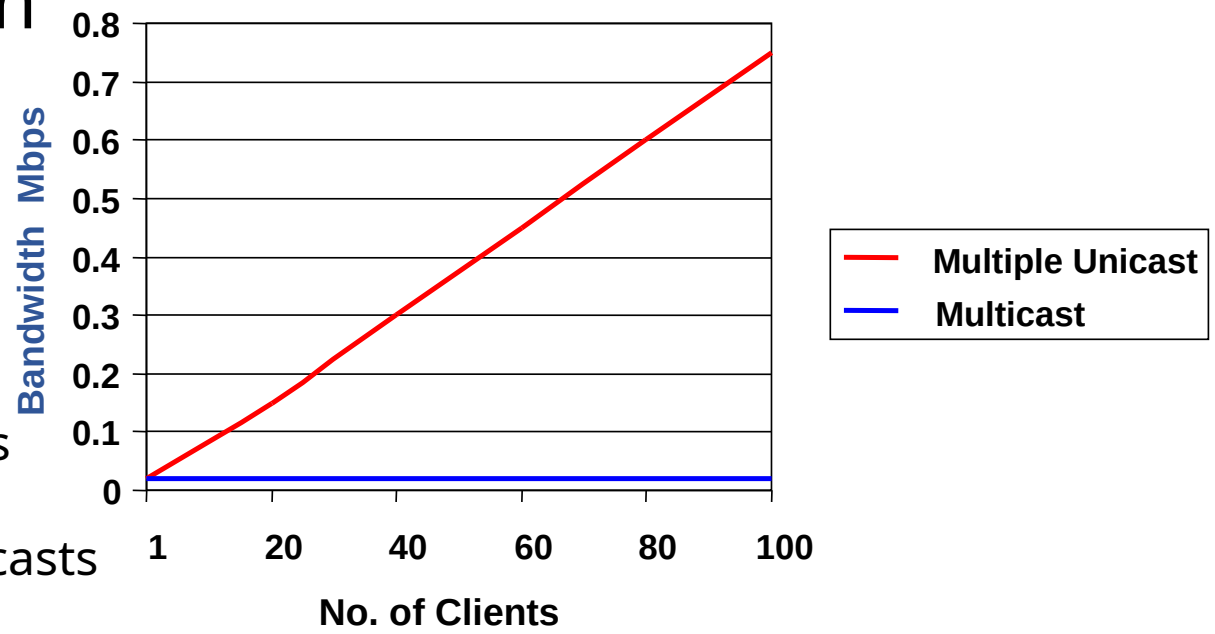
- This enables the **multicast** of a message to a **group of processes** as a **single action**
  - With no need to list the set of individual destinations of the messages that we send to the group (this is managed for us by the middleware)

- This supports the efficient dissemination of data, with uses including service discovery, publish/subscribe, replication, and shared channels

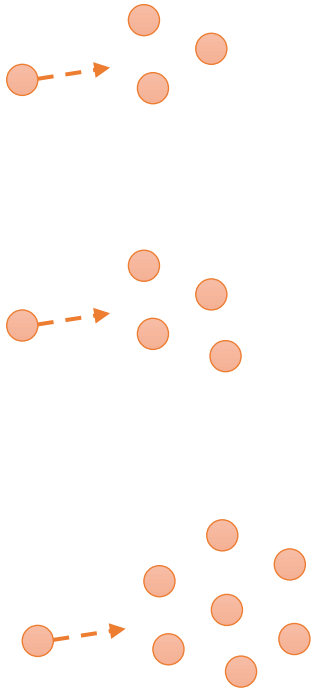
Associated reading: TvS Section 7.4; CDKB Sections 4.4 & 6.2

If we had network-level multicast support...

- Bandwidth profile would look something like this, with increasing members of a group in a one-to-many communication



The server only sends a single message, and overall network traffic is reduced whenever parts of a route are the same, compared to multiple unicasts

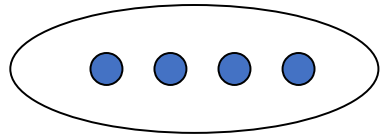


Unfortunately, in general, we don’t have network-level support :(

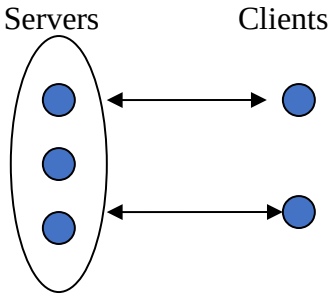
- This means that we have to build multicast, or group communication, at the application level

- In practice this really means a sender is going to send an identical message to each member of the group, as a set of unicast communications

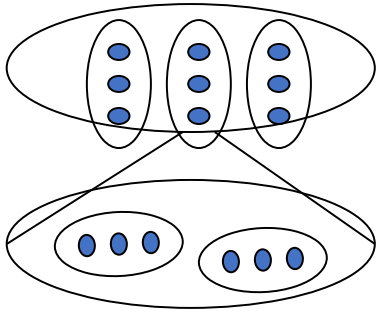
Types of group communication



- Membership styles:
  - Open (anyone can join)
  - Closed (invitation)



- Structure
  - Hierarchical
  - Peer-to-peer
  - Client-server / replicated servers

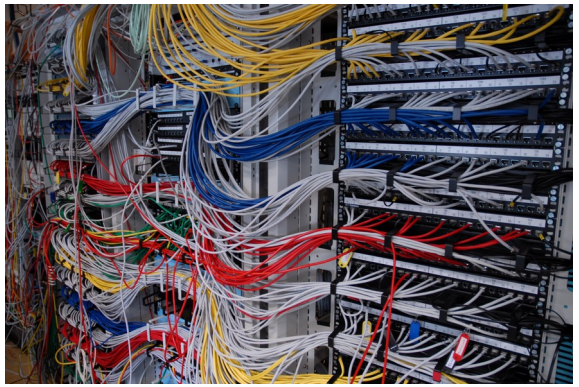


Common applications

Live video streaming (1-many)

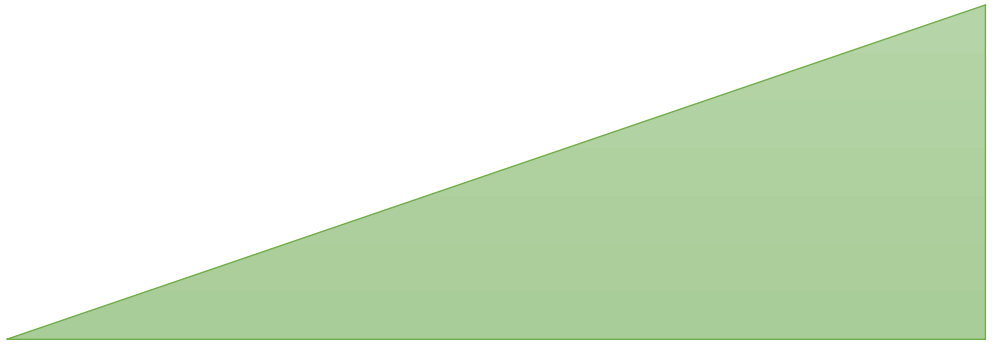
Air traffic control (many-many)

Multi-player gaming (many-many)



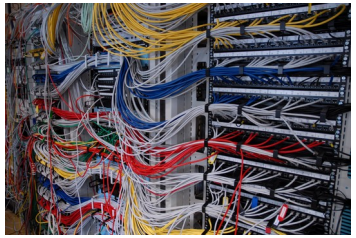
Server Replication (1-many, many-many)

How much reliability do I need...?



Best effort?

Atomic?



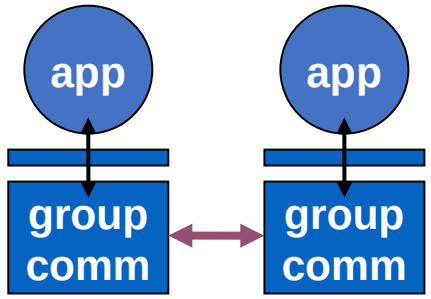
Missing a video frame / game state change is a little annoying...

Missing an update is a potential risk to life, or can cause money to be "lost in transit"

Not everything is possible...

- In distributed systems we have to work within the bounds of what is theoretically known to be possible; we have well-known formal proofs of things that are *not* possible

A typical group communication service...




```
interface GroupCommunicationService {

    // creates a new group and returns the groups ID

    public GroupID      groupCreate();

    // Adding & Removing a member to/from a group

    public void         groupJoin   (GroupID group, Participant member);
    public void         groupLeave  (GroupID group, Participant member);

    // multicasts a message to the named group with
    // the specified delivery semantics, and
    // optionally collects a number of replies

    public Messages[ ] multicast    (GroupID group, OrderType order,
                                     Messages message, int nbReplies)

}
```

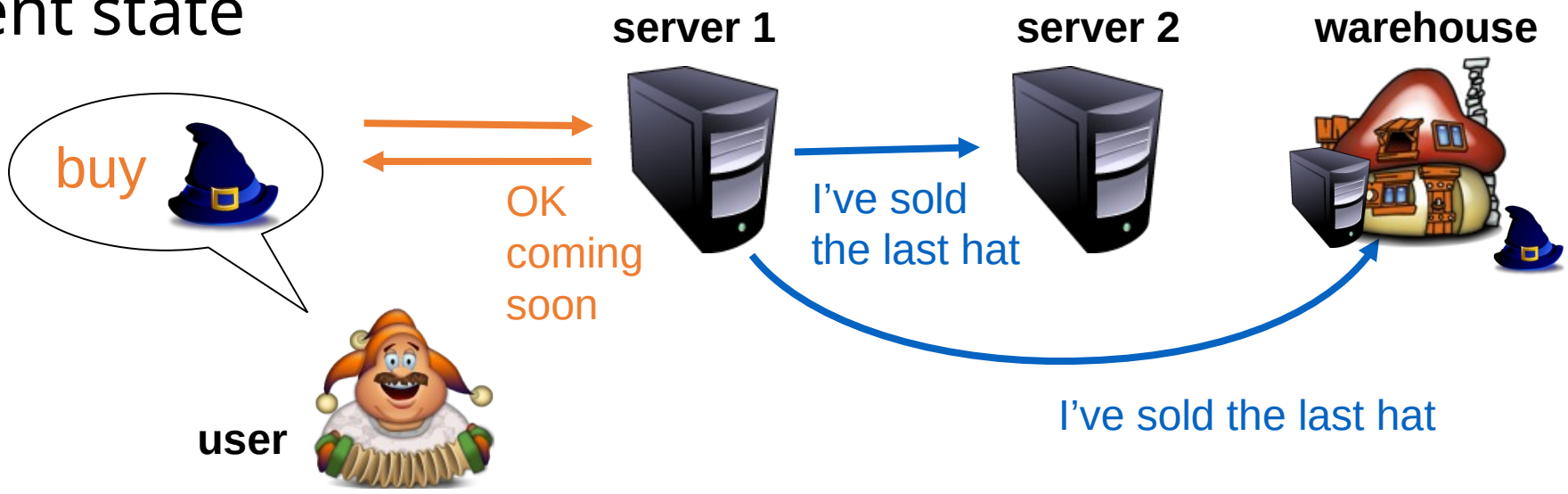

How that really works...

- We would advertise a group service at a particular endpoint, often a URL such as www.google.com/groups/xyhfs

- A program listens for communication with this URL and directs users to the group service
  - (this is more or less how Discord works)

Example: active replication

- When we replicate a server (either for *fault-tolerance* or for *availability*) we need collective communication between those servers to ensure that clients and servers observe a consistent state



Challenges

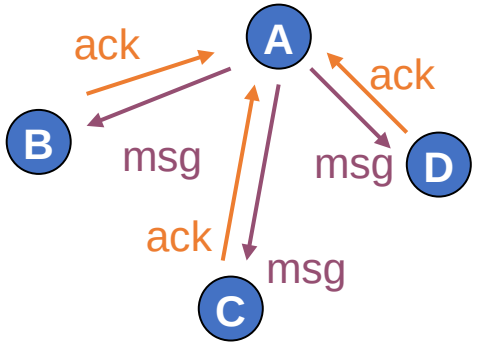
- Reliability?
- Scalability?
- Ordering?
- Coordination?

Reliability guarantees

- **Best effort:** Unreliable multicast
  - No guarantee
  - Message sent to all members and may or may not arrive
- **Reliable multicast:** Protection against faulty network
  - Reasonable efforts are made to ensure delivery in spite of message losses
  - No guarantees if the sender crashes during multicast
- **Atomic multicast:** Protection against faulty participants
  - All members receive message, or none does
  - Main issue: tolerate the sender's crash

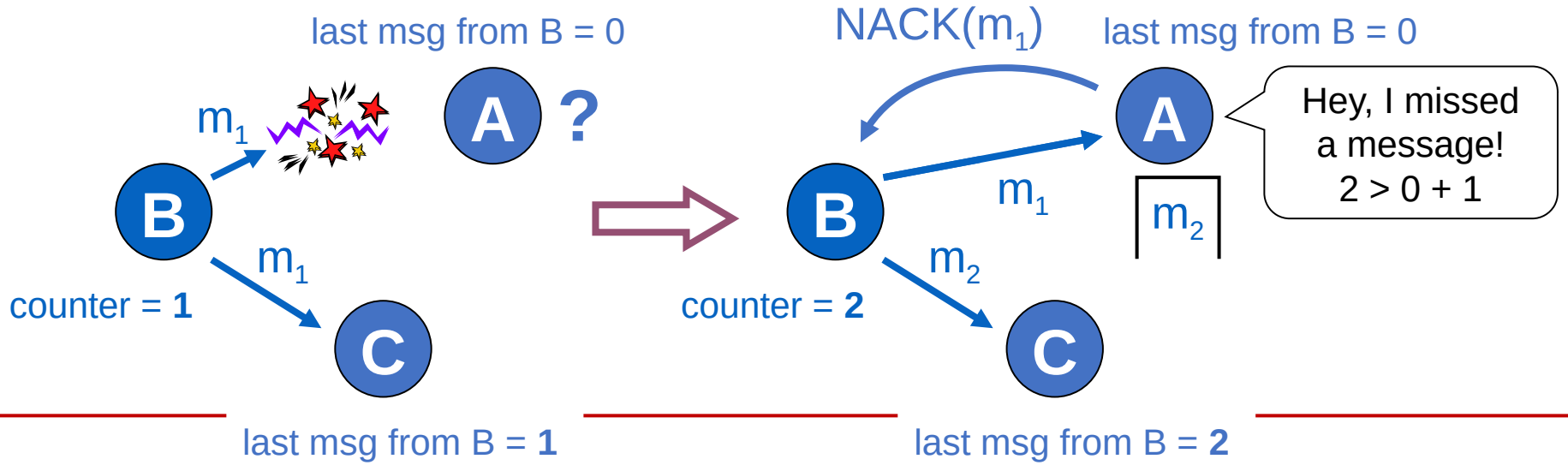
Implementing Reliable Multicast

- A message is sent to each member of the group, and acknowledgements (ACK) are awaited
- If some acknowledgements aren't received in a given time period, re-send message; repeat this n times if necessary
- If all acknowledgements received
  - report success to caller
- Works fine if:
  - Network problems are transient (messages eventually get through)
  - No crash (and no spurious behaviour)
- Not very scalable: ACK explosion!



Avoiding ACK explosion

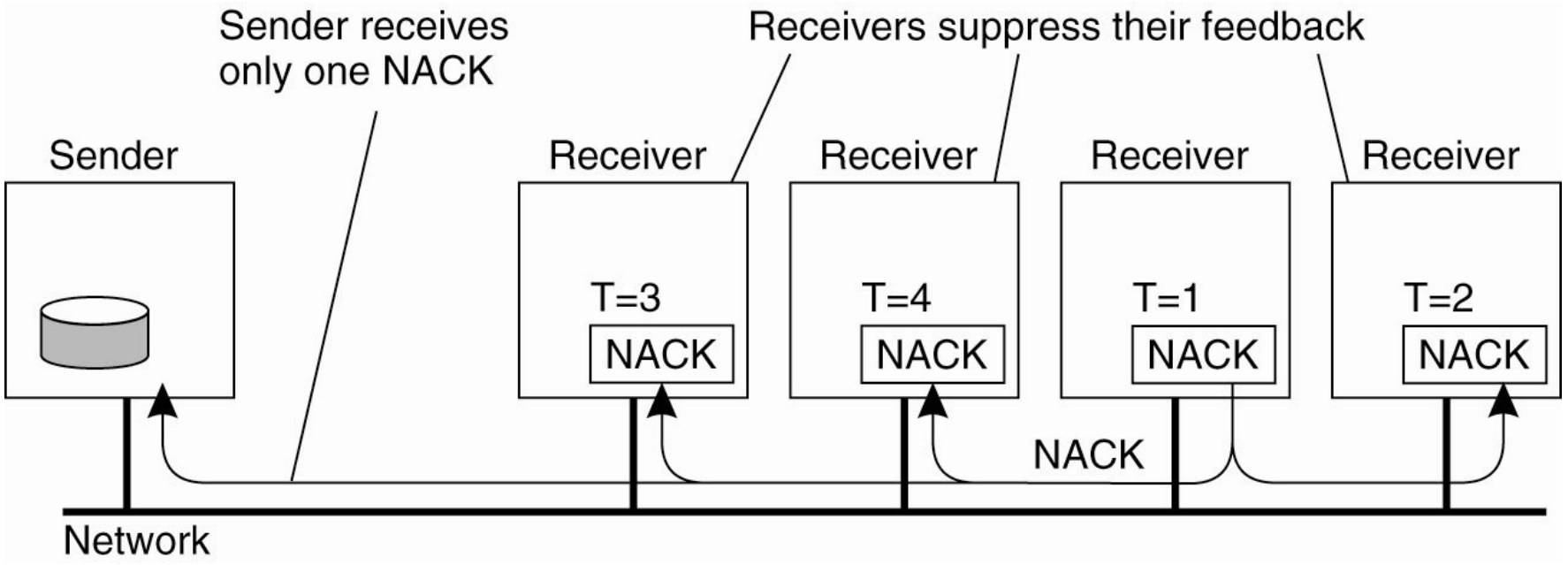
- Using Negative ACKs (NACKs)
  - If everything is fine the receiver does not say anything
  - If a message is lost the receiver complains to the sender
- Problem: How do we know that a message should be there?



Negative ACK notes

- **ACK explosion** is avoided
  - A NACK explosion might still occur but is less likely
- **Garbage collection** problem
  - In theory sender should keep all past messages indefinitely
  - In practice past messages only kept for a “long enough” period
- More advanced schemes possible
  - Limiting NACK instances using **NACK suppression**
  - **Hierarchical** Feedback Control

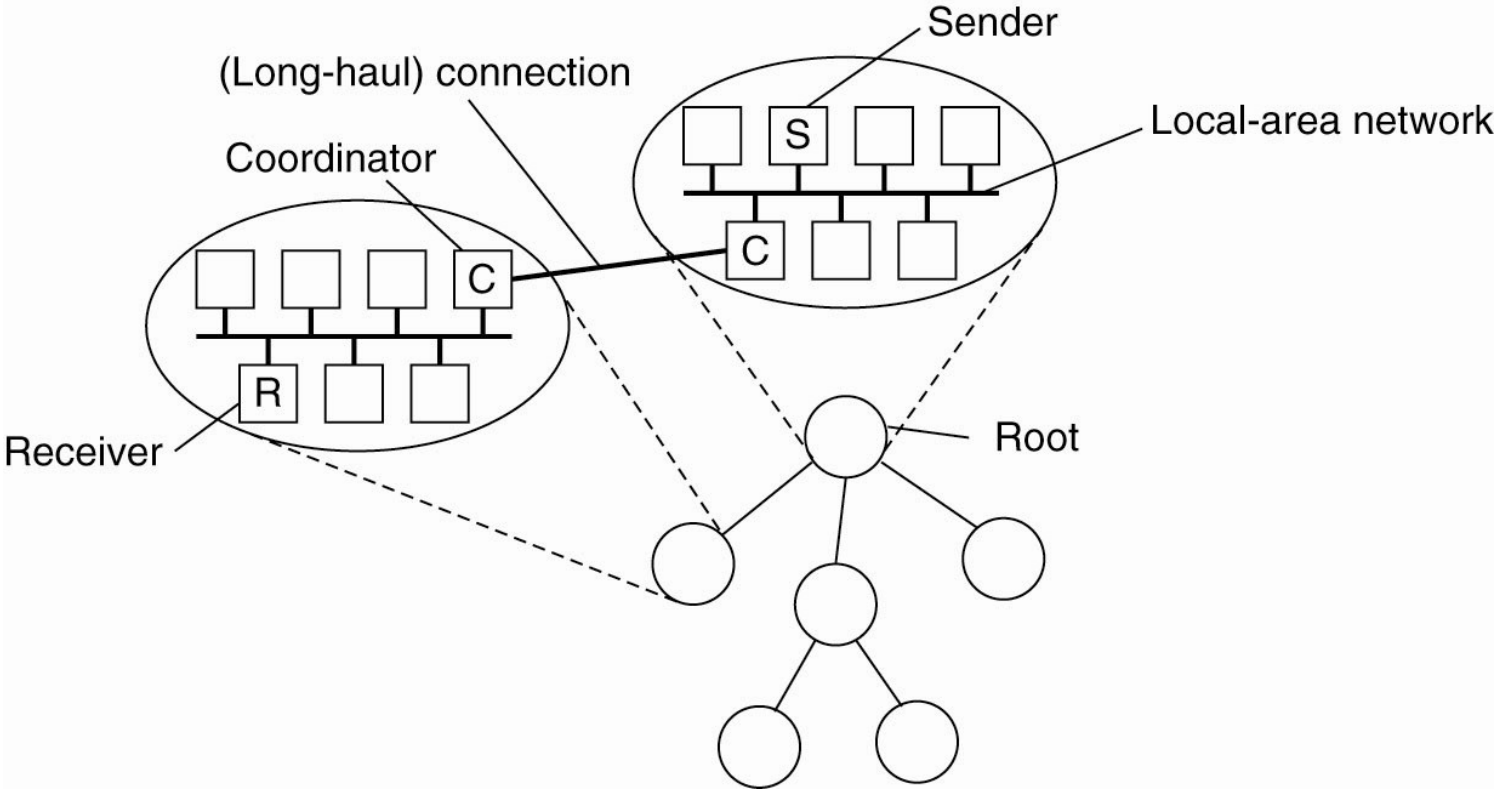
NACK suppression



Several receivers schedule a request for retransmission, but the first retransmission request leads to the suppression of others.

Associated reading: RFC 5401; TvS Section 8.4.2

Hierarchical feedback control

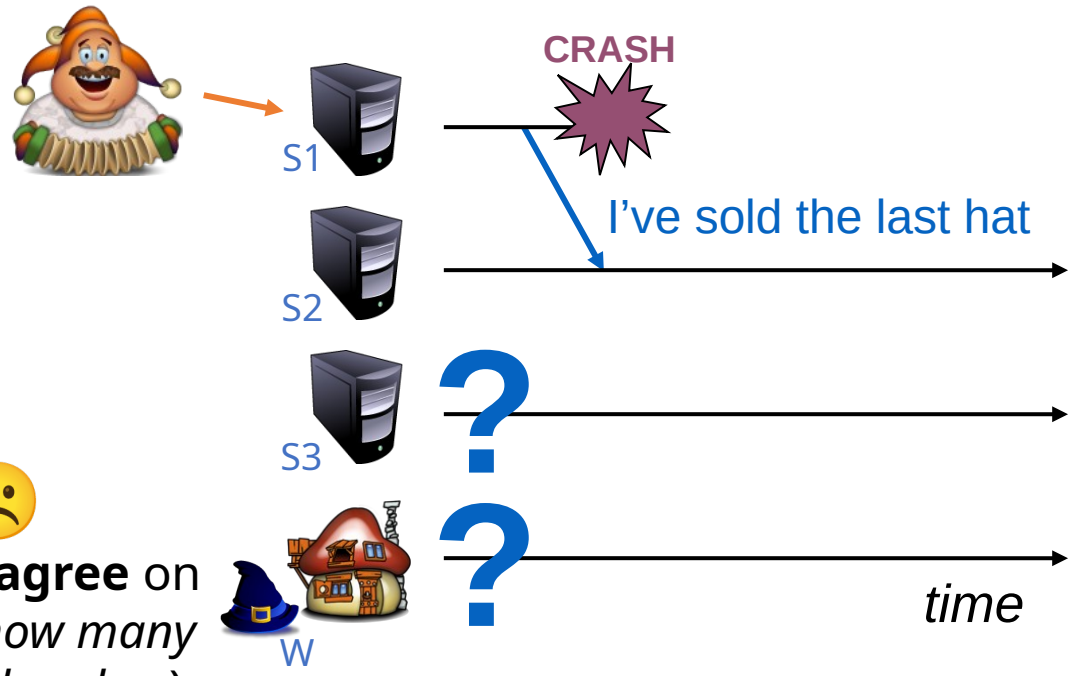


Each local coordinator forwards the message to its children and later handles retransmission requests.

Associated reading: TvS Section 8.4.2

Death of a sender...

- Reliable broadcast caters for *network* problems...but what if *participants* fail?



The customer never gets their hat ☹

Worst case: the surviving servers **disagree** on what has happened! (*nobody knows how many hats are left, or how much money the shop has*)

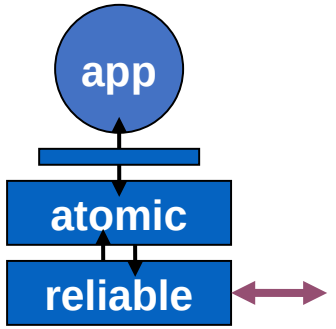
Atomic Multicast

Atomic Multicast

- We need a stronger multicasting mechanism
  - Either everybody in the group gets the message
  - Or nobody (who's surviving) does

- All or nothing: Atomic (i.e. which can't be cut)

- Atomic multicast protocol
  - Sender: reliably multicasts messages to rest of group
  - All participants:
    On receiving message:
    - If first time I'm seeing this message then {
      reliably multicast again;
      and deliver to application; }
    - else { discard copy ; }



Atomic Multicast

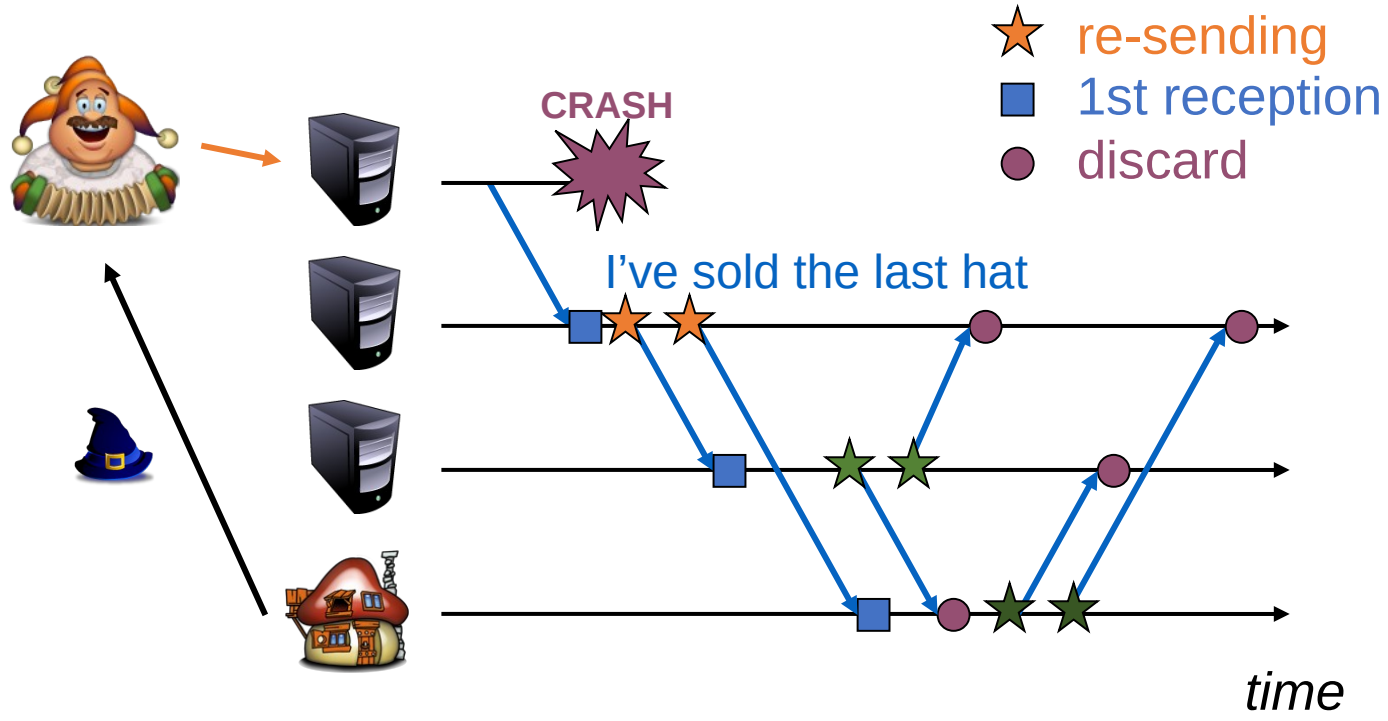
Sender's program

```
i:=0;
do   i ≠ n
   reliable_multicast(m,member[i]) ;
    i:= i+1                                  else

   duplicate
```

Receiver's program

```
 if message m is new
     reliable_multicast  m;
     accept m;


      discard    m;    //    m    is
```


Atomic Multicast: everyone gets the message



The cost of this is far more messages sent; there are different protocols which try to optimise this

Associated reading: TvS Section 8.4.3

Summary

- Group communication is another very common middleware layer for distributed communications

- As we saw in remote invocation, we again have a number of different levels of reliability that we can gain depending on what our particular application (or sub-element of that application) needs – but the higher the level of guarantee that we want, the more we have to pay in network costs or latency