Exam and Term Project

- Exam
  - Exam, Monday, April 8, 2024
  - **Time: 3:15pm – 5:15pm; Room SB111**
  - Close book, Close note, no phone, calculator, or computer or any internet access

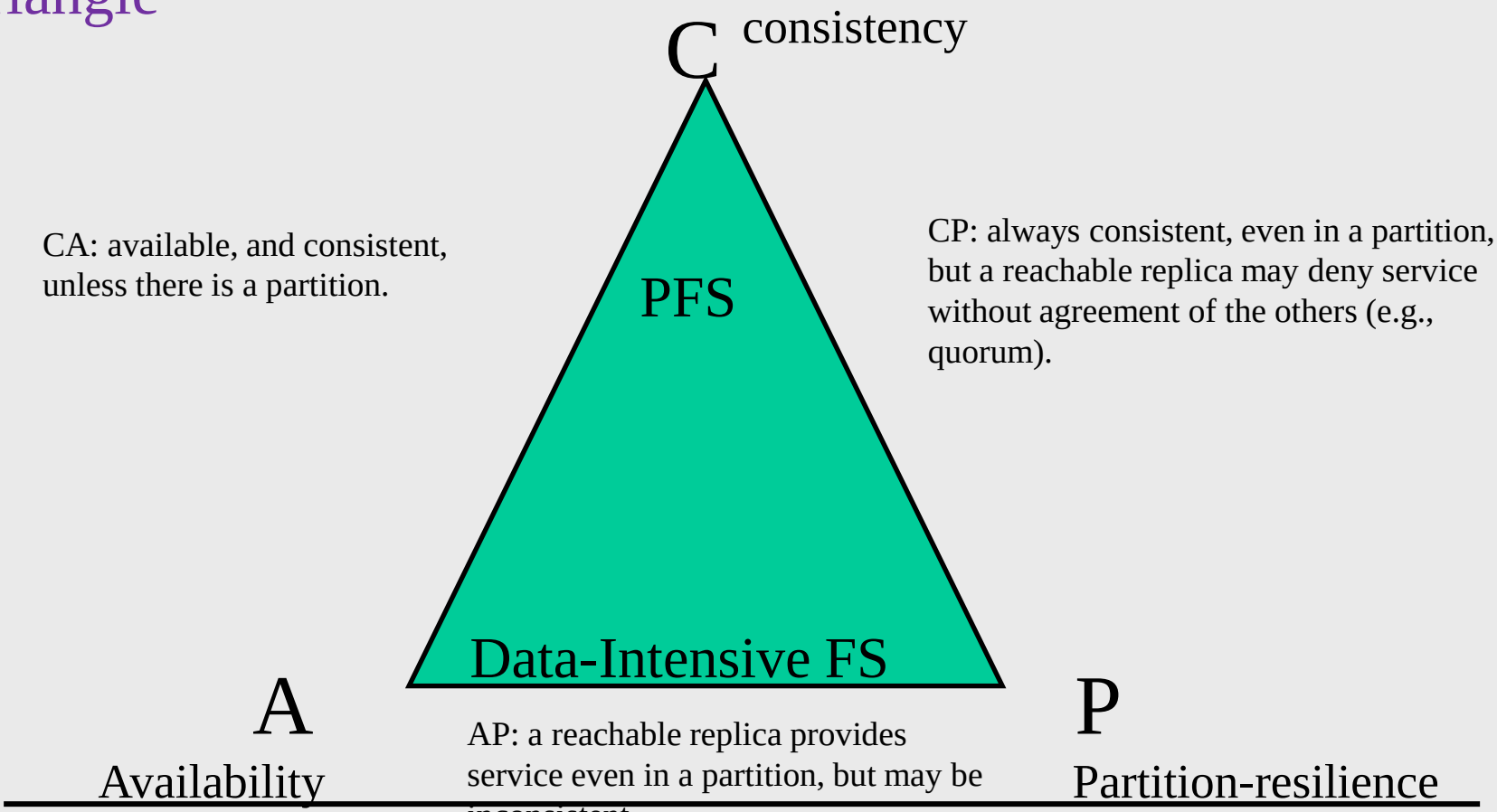
- Term Project
  - Presentation, Monday and Tuesday, April 22
  - Term project report, April 25, 2024
  - Progress report March 20

Guest Lecture After Spring Break

- Wednesday
  - Dr. Bogdan Nicolae
  - AI, Deep Learning, and HPC

The CAP Theory: only Two/Three

Claim: Every distributed system is on one side of the triangle



Consistency, Availability, & Partitioning

CAP Theorem: Any networked system providing shared data can provide only two of the following three properties:

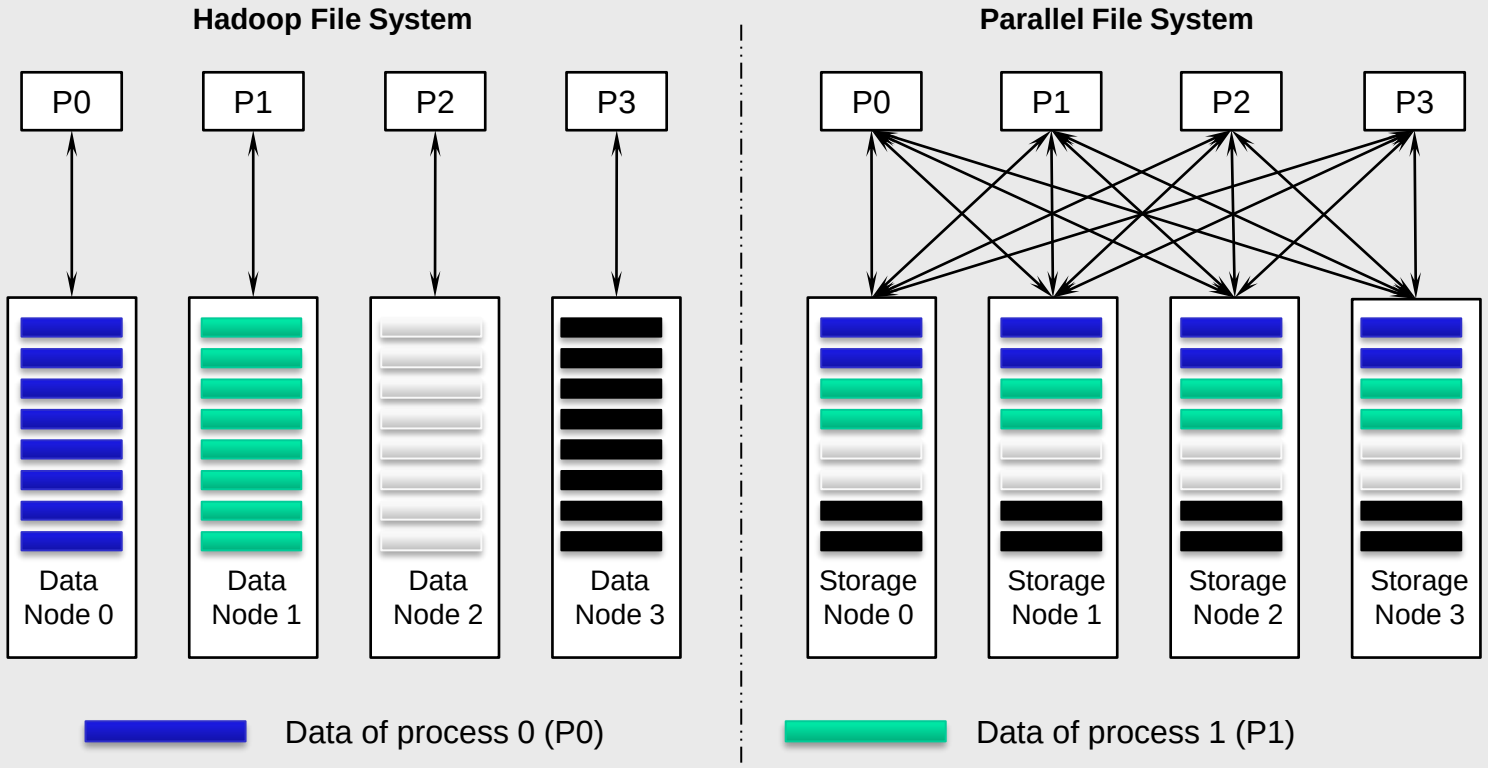
- **C**: consistency, by which a shared and replicated data item appears as a single, up-to-date copy
- **A**: availability, by which updates will always be eventually executed
- **P**: Tolerant to the partitioning of process group (e.g. because of a failing network)

• In other words, in a network subject to communication failures, it is impossible to realize an atomic read/write shared memory that guarantees a response to every request

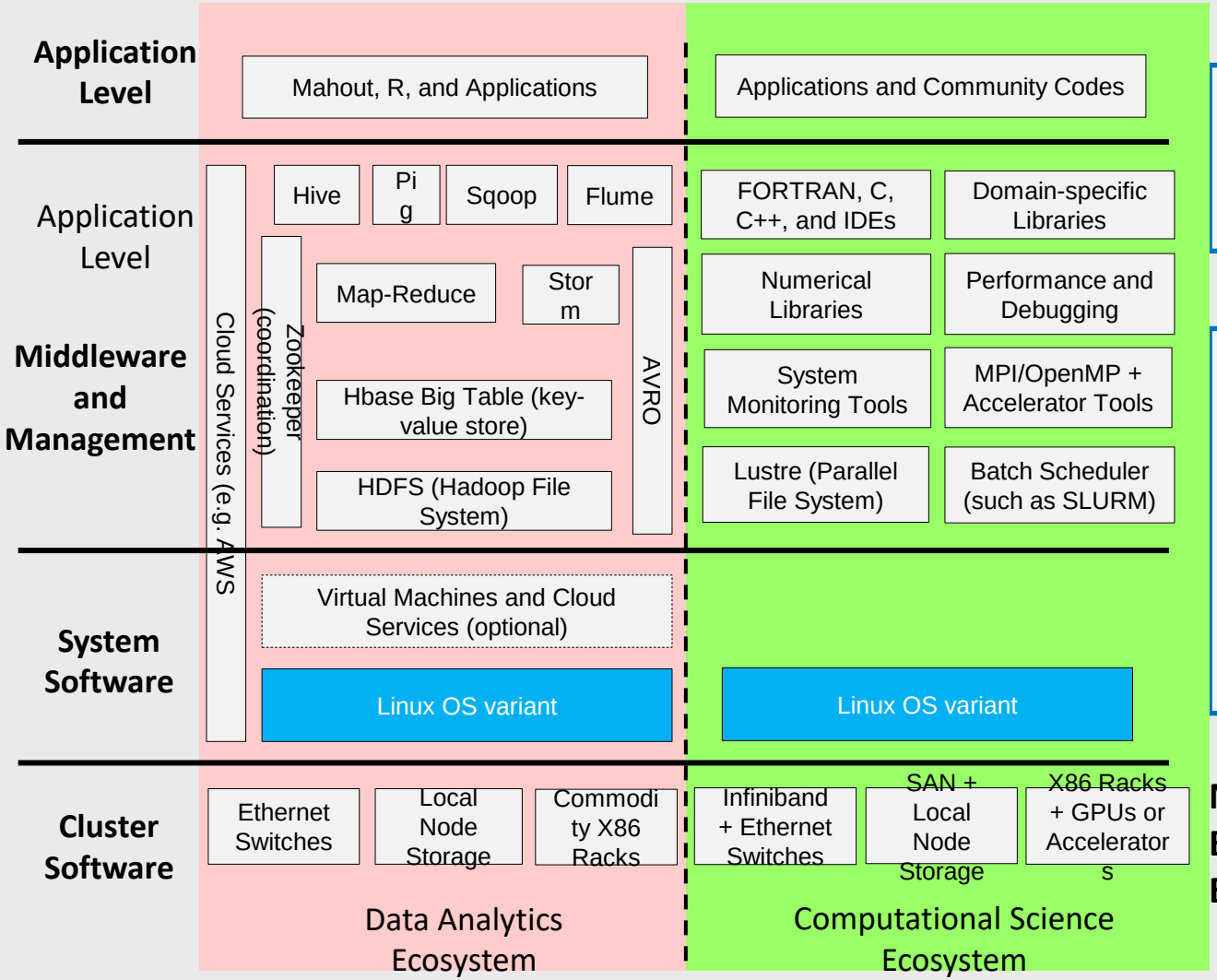
Eric Brewer 2000, Gilbert and Lynch 2002

Two Camps of High Performance File Systems

- Google Hadoop File System (HFS): Perfect parallel
- IBM Parallel File System (PFS): Consistent
- The two do not talk to each other



The Data & Computing Ecosystems



Computational Science

FORTRAN,C,C++: languages
PAPI: performance and debugging tool
MPI/OpenMP: multi-core parallel model
SLURM: batch scheduler
Lustre: parallel file system

Data Analytic

Mahout: machine learning tool
Hive: data warehouse software
Pig: provide high level language for big data
Sqoop: exchange data with traditional database
Flume: log management
Zookeeper: maintaining consistency
Storm: real-time computation system.
Hbase: a distributed, scalable big data store.
AVRO: data serialization system.

NOTE: The Divergence of Big Data and HPC Eco-Systems!

Courtesy by "Exscale Computing and Big Data", DANIEL A. REED AND JACK DONGARRA , CACM July 2015

Chapter 8: Fault Tolerance

- Basic concepts in fault tolerance
- Masking failure by redundancy
- Process resilience
- Reliable Client-Server communication
  - One-one communication
  - RPC semantics
- Reliable Group Communication
- Distributed commit
- Failure recovery
  - Checkpointing
  - Message logging

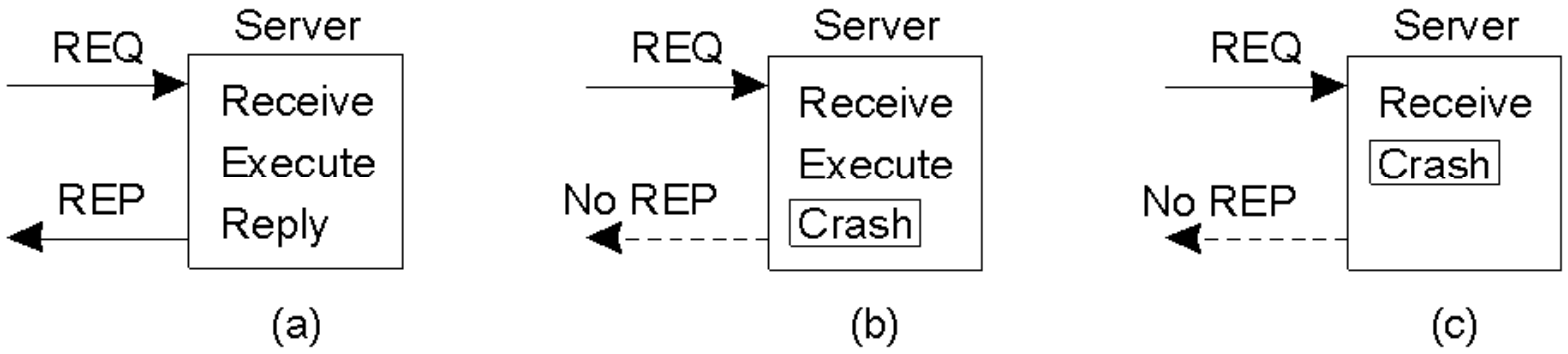
Reliable One-One Communication

- Make use of a reliable transport protocol (TCP) or handle at the application layer
- Five different classes of failures in RPC systems:
  - Client unable to locate server
  - Lost request messages
  - Server crashes after receiving request
  - Lost reply messages
  - Client crashes after sending request

Reliable One-One Communication

- Client unable to locate server
  - Updating, exception (*Sig-NoServer*)
- Lost request messages
  - Repeat, cannot locate sever
- Server crashes after receiving request
  - At least once (keep trying), at most once (give up), guarantee nothing
- Lost Reply Message
  - Idempotent request, sequence number
- Client Crashes
  - Orphan, Extermination, Reincarnation, Gentle Reincarnation, Expiration
  - Termination may not be desirable

Lost Request Messages Server Crashes



- A server in client-server communication

a) Normal case

b) Crash after execution

c) Crash before execution

Server Crash

- Assume client requests server to print a msg
  - Send a completion msg (M) before print (P), or
  - Send a completion msg (M) after print (P)
- Combinations
  - M→P→C: crash after ack and print
  - M→C(→P):
  - P→M→C:
  - P→C(→M)
  - C(→P→M): crash before print and ack
  - C(→M→P)

- When a crashed server recovers, the client can
  - never reissue a request
  - always reissue a request
  - reissue a request only if it received ack
  - reissue a request only if it received no ack

No perfect solution

Client	Server					
	Strategy M -> P			Strategy P -> M		
Reissue strategy	MPC	MC(P)	C(MP)	PMC	PC(M)	C(PM)
Always	DUP	OK	OK	DUP	DUP	OK
Never	OK	ZERO	ZERO	OK	OK	ZERO
Only when ACKed	DUP	OK	ZERO	DUP	OK	ZERO
Only when not ACKed	OK	ZERO	OK	OK	DUP	OK

ok: Text is printed once; dup: printed twice; zero: no printout

Reliable One-One Communication

- Lost Reply Messages
  - Some requests can be re-executed without harm (idempotent) (e.g. Read 1024 bytes of a file); some not.
  - Solutions:
    - Structure requests in an idempotent way
    - Assign request a sequence number to be checked by server
- Client Crashes leading to orphan computation
  - extermination: client side logging of RPC about what to do; the log is checked after a reboot.
  - reincarnation: client bcasts a new epoch when it reboots; server detects orphan computations based on epochs.
    - kill orphan remote computation or locate their owners
  - expiration: set a time quantum for each RPC request; if it cannot finish, more quanta are asked.

Group Communication

- Communicate to a group of processes rather than point-to-point
- Applications:
  - Fault tolerance based on replicated service
  - Locating objects in distributed services
  - Better performance through replicated data (caching)
  - Efficient dissemination of “news”
    - pushing technology: weather, stocks, etc
  - Building block for protocols:
    - memory consistency, etc.

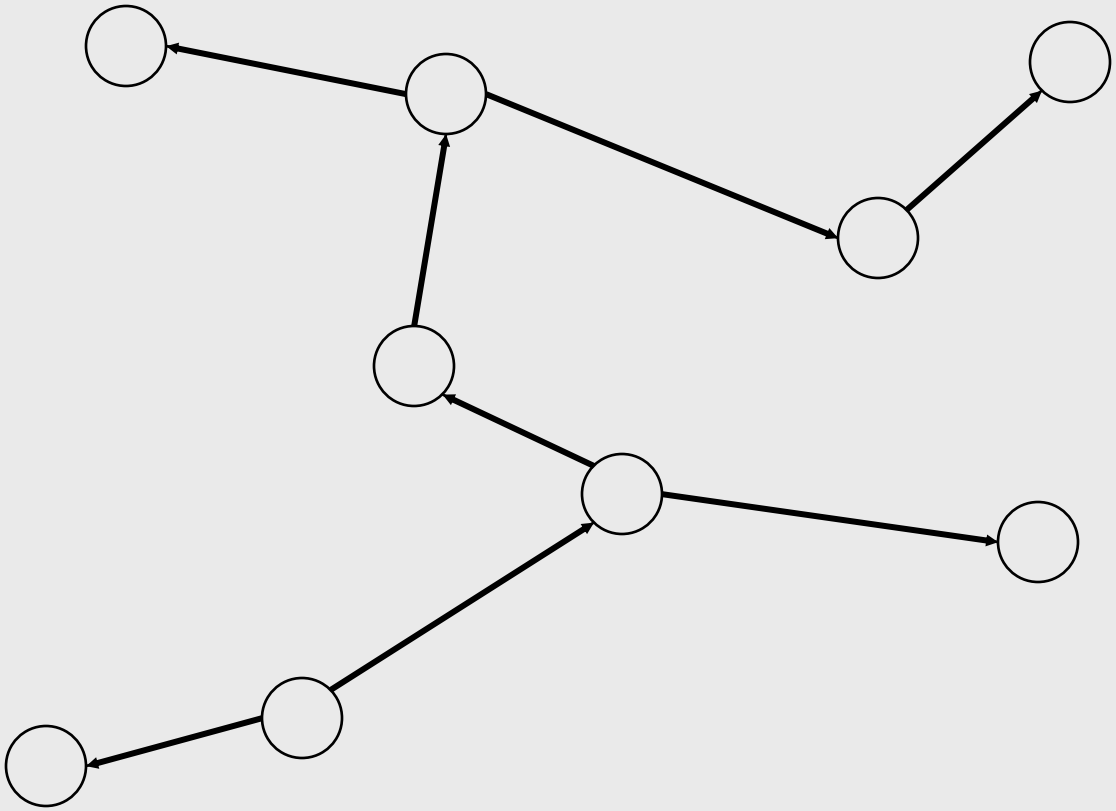
Example: Replicated File Servers

- N copies of file server for tolerating failures of some servers
  - N may vary over time
- send commands to all servers
- all servers execute commands
- want to keep all servers in identical states
  - make sure that all replicas get the same sequence of commands.

Why Group Comm. (E.g. Multicast)

- Convenient programming
  - Group Comm API
- Efficiency
  - Efficiently utilize network bandwidth
  - Reduce the total time to deliver a msg to a group of processes
- Delivery guarantee
  - Guarantee of atomic delivery
    - For a msg multicasting, it is either delivered to all processes as a while or none
  - Guarantee of delivery ordering

Multicast Trees



Design spectrum

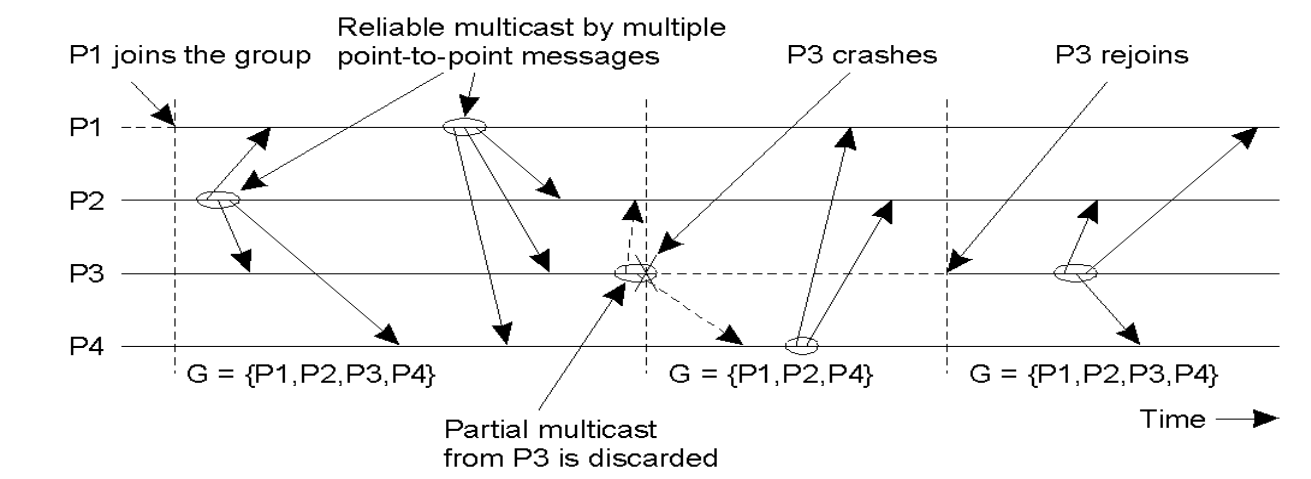
- Simplest: unreliable multicast without guarantees about delivery or ordering
- Complex: atomic and in-order multicast.
  - Atomic: either received by all or none.
  - Ordering: ordered delivery of messages from different senders
  - Reliable multicast in the presence of failures
- Compromise in applications:
  - trade off speed and complexity vs. ease of use

Groups

- a group is any set of processes that want to cooperate; processes may fail only by crashing
- processes can join or leave groups, either by explicit action or implicitly
- a process may belong to several groups
- basic operation of group communication: multicast a message to an entire group
  - mcast(g, m): sends the msg m to the group g
  - deliver(m): delivers the msg sent by mcast to the calling process
- group name provides a useful level of indirection

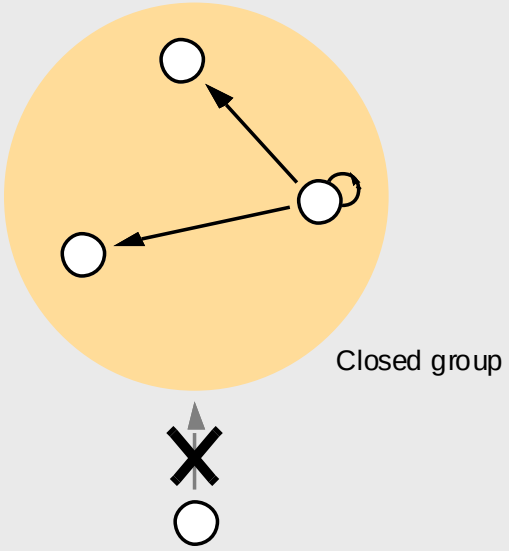
Atomic Multicast

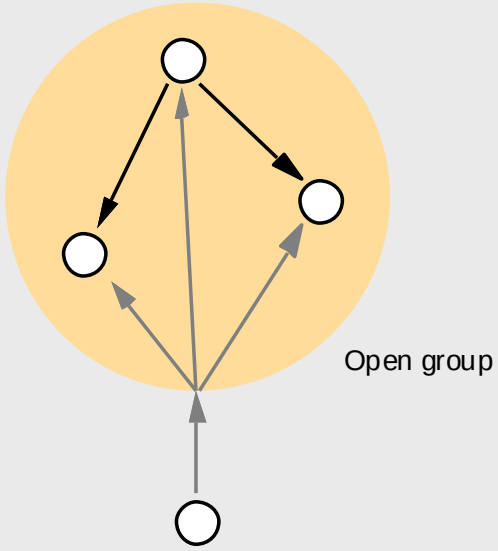
- **Atomic multicast**: all processes received the message or none at all
- Solution: **Group view & View change**
  - Each msg is uniquely associated with a group of procs
  - View of the process group when message was sent
  - All procs in the group should have the same view
  - Virtually synch property



Closed vs. Open Groups

- Open group: anybody may send a message to the group
- Closed group: only a member may send messages to the group. A process in a closed group delivers to itself any msg that it multicasts to the group.
  - security implications
  - usually more efficient, since all senders know who is in the group
- Master-slave group: only the “master process” can send to the group

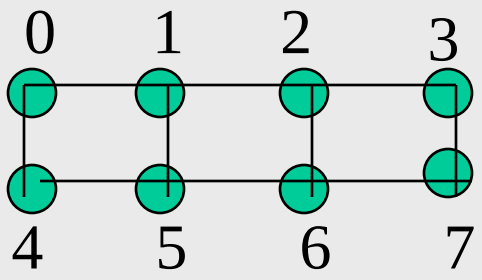




Case 1: Group Communication in MPI

- All MPI communication is relative to a ***communicator*** (group and context***)***
  - e.g. MPI_COMM_WORLD
  - int MPI_Barrier(MPI_Comm comm)
  - int MPI_Bcase(void* msg, int count, MPI_Datatype dtype, int root, MPI_Comm comm)
- Create a new communicator
  - Assume MPI_COMM_WORLD has p processes, where p=q^2, which are inter-communicated in a grid
  - Create a communicator from processes in the first row

Create a communicator




```
MPI_Group       MPI_GROUP_WORLD
MPI_Group       first_row_group;
MPI_Comm        first_row_comm;
int size;
int * proc_ranks = (int *)malloc(q*sizeof(int));

for(proc=0; proc<q; proc++)
      proc_ranks[proc] = proc;
MPI_Comm_group(MPI_COMM_WORLD,
                                &MPI_GROUP_WORLD);
MPI_Group_incl(MPI_GROUP_WORLD, q, proc_ranks,
                                &first_row_group);
MPI_Comm_create(MPI_COMM_WORLD, first_row_group,
                                &first_row_comm);
```

Case 2: IP Multicasting on the Internet

- IP mcast allows the sender to transmit a sngle IP packet to a set of hosts that form a multicast group.
- A multicast group is specified by a class D Internet addresses: group D begins with 1110
  - from 224.0.0.1 to 239.255.255.255
  - Dynamic membership management.
- Multicast requirements
  - TCP/IP protocol be multicast enabled. Available only via UDP
  - Routers be multicast capable
  - Ethernet switches support multicast
- Permanent group addresses: 224.0.0.1 to 224.0.0.255
  - ftp://ftp.isi.edu/in-notes/iana/assignments/multicast-address
  - ALL-SYSTEMS.MCAST.NET 224.0.0.1
  - ALL-ROUTERS.MCAST.NET 224.0.0.2
  - NTP.MCAST.NET 224.0.1.1

Sender vs Receiver

- Multicast receiver
  - Get a UDP socket
  - Bind to the application's port number
  - Join the application's multicast address
  - Receive
  - Close the socket when complete
- Multicast sender
  - Get a UDP socket
  - Set the IP Time-To-Live appropriately
  - Send to the application's multicast (addr, port)
  - Close the socket when complete
- Java API: MutlicastSocket as a subclass of DatagramSocket

Time-To-Live (TTL)

- TTL controls the coverage of a packet
  - Local host 0
  - Local subnet 1
  - Local campus 16
  - United States 48
  - North America 64
  - High-bandwidth worldwide 128
  - All sites worldwide 256
- Java API
  - setTimeToLive()
  - By default, TTL=1: propogate only on the local net

Case 3: Multicasting in Java: Sender

```
// java MulticastSender ALL-SYSTEMS.MCAST.NET 4000
public class MulticastSender {
      public static void main( String[] argv ) {
            InetAddress ia = InetAddress.getByName( argv[0] );
            int recvPort = Integer.parseInt( argv[1] );
            try {
                  String str = “Hello from”+InetAddress.getLocalHost().toString;
                  byte[] data = str.getBytes();
                  DatagramPacket dp =
                        new DatagramPacket(data, data.length, ia, recvPort);
                  MulticastSocket ms = new MulticastSocket( recvPort );
                  ms.joinGroup(ia);            //Join a multicast group
                  ms.send(dp, (byte)1));       // Time-To-Live (TTL)=1
                  ms.leaveGroup(ia);
                  ms.close();
            } catch (IOException e) {}
}
```

Multicasting in Java: Receiver

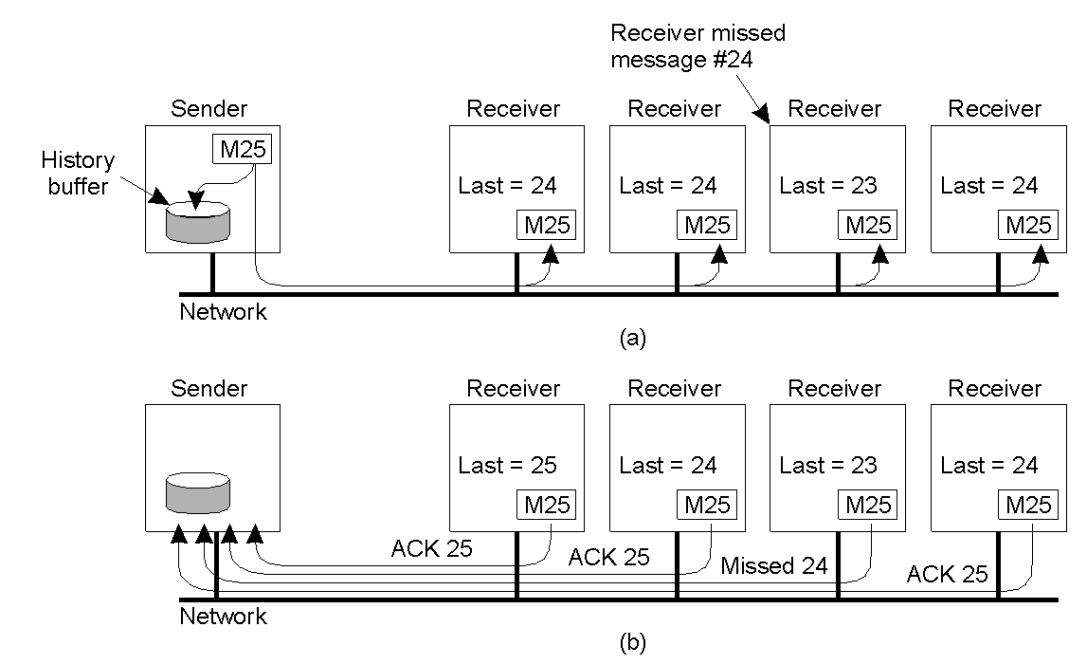
```
// java MulticastSnooper ALL-SYSTEMS.MCAST.NET 4000

public class MulticastSnopper{
      public static void main( String[] argv ) {
            InetAddress ia = InetAddress.getByName( argv[0] );
            int port = Integer.parseInt( argv[1] );
            DatagramPacket dp =
                        new DatagramPacket(new byte[1024], 1024);
            try {
                  MulticastSocket ms = new MulticastSocket( port );
                  ms.joinGroup(ia);            //Join a multicast group
                  ms.receive(dp);
                  System.out.println( new String(dp.getData() );
                  ms.leaveGroup(ia);
                  ms.close();
            } catch (IOException e) {}
}
```

Best-Effort Multicast: Drawbacks

- Not reliable, some members might miss some messages
- different members see different messages arriving in different orders
  - causes trouble for replicated services
    - example: replicate file servers with “create foo” and “delete foo” messages
- different members may see different group membership at the same time

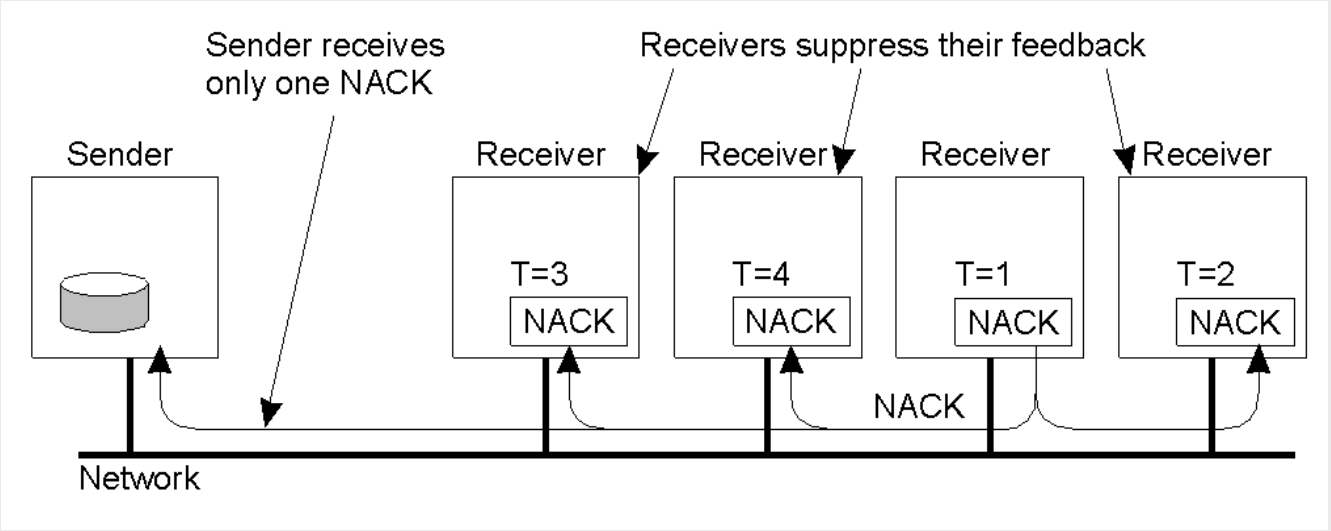
Simple Reliable-Multicast Schemes



- Multithreading to improve efficiency?
- Ack-impulsion issue: Acks of the reliable send ops are liable to arrive at about the same time. These acks will quickly fill the mcasting buffer and some acks would be dropped. Msg retransmissions lead to more acks.
- Negative acknowledgements only?
  - Msg needs to be buffered at the sender side forever (in theory)

Scalable Multicast (I)

- Feedback suppression
  - Negative acknowledgement only
  - Feedback be mcasted to the rest of the group, after some random delay
  - Feedback can be suppressed by another group member so as to ensure only single request for retransmission reaches the sender

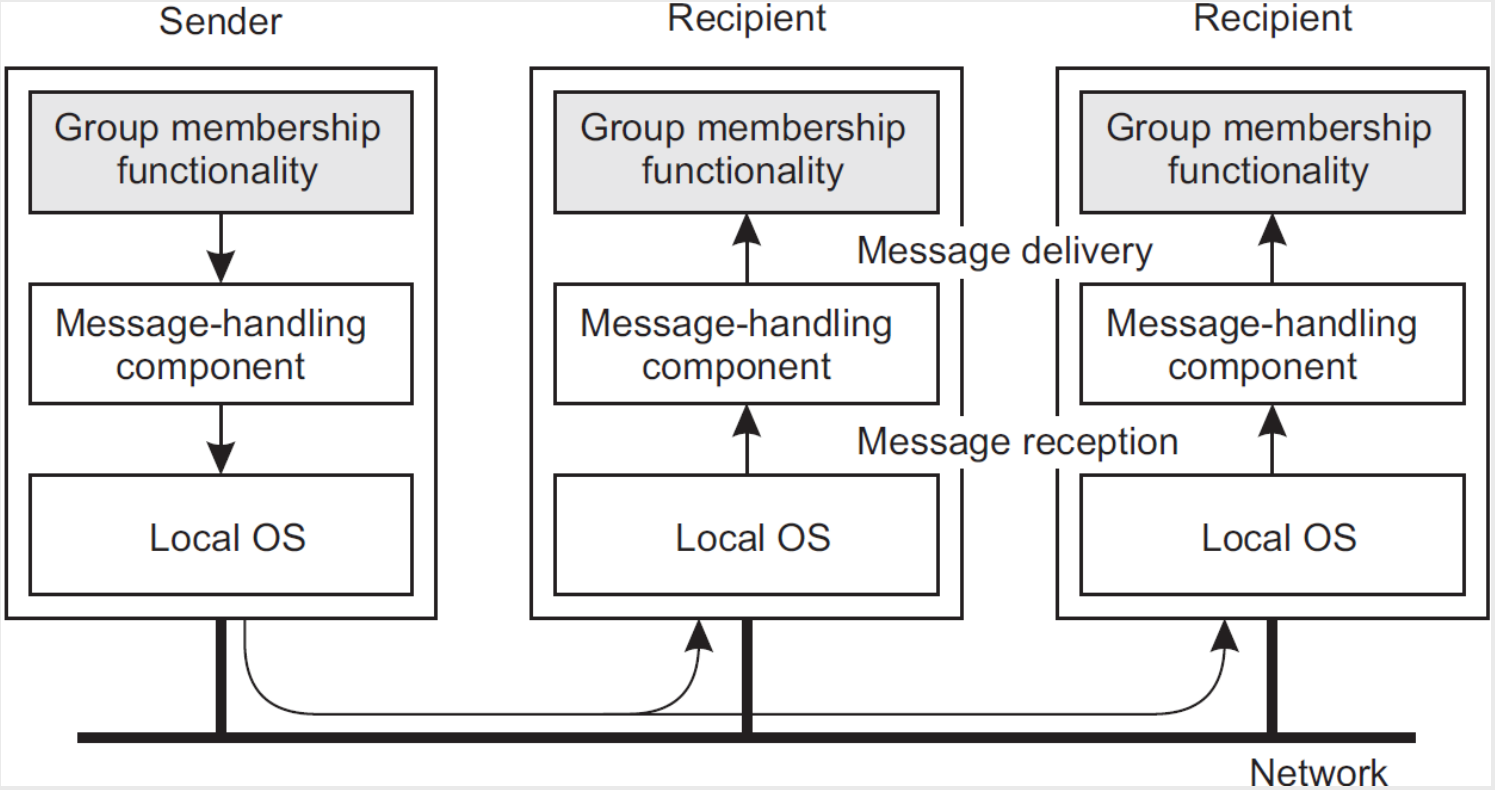


Disadvantages:

- Complex feedback message scheduling (delay?)
- Interruption of successful processes

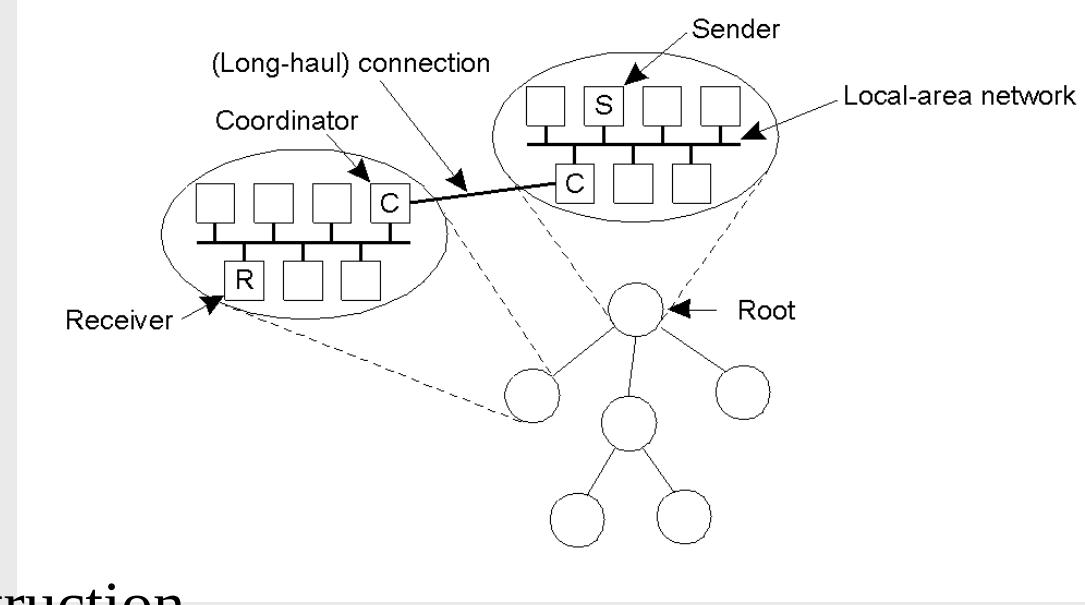
Sender and Receiver

- Message and Group membership
  - May need to forward the message



Scalable Multicast (II)

- Hierarchical feedback control
  - Group of receivers is recursively partitioned into a number of subgroups, which form a tree structure
  - Subgroup containing the sender serves as the root
  - Each subgroup designates a local coordinate for handling retransmission requests of receivers contained in the subgroup
    - maintain its own history buffer



Challenge is
fast dynamic tree construction

Reliable Multicast

- Basic properties:
  - Validity: If a correct process multicasts message m, then it will eventually deliver m.
  - Integrity: a correct process delivers the msg at most once
- Atomic messages (aka agreement)
  - A message is delivered to all members of a group, or to none
- Message ordering guarantees
  - within group
  - across groups
- Membership agreement guarantees in the presence of failures, w.r.t. msg ordering

Message Ordering

- Different members may see messages in different orders
- Ordered group communication requires that all members agree about the order of messages
- Within each group, assign *global ordering* to messages
- Hold back messages that arrive out of order (delay their delivery)

Message Ordering

- FIFO ordered multicast
  - Deliver incoming messages from the **same process** in the same order as they have been sent
- Causally ordered multicast
  - If two message causally related, then the multicast should keep their order
- Total-ordered multicast
  - When message is delivered, they are delivered in the same order to all group members