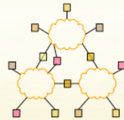
15-446 Distributed Systems Spring 2009



L-15 Fault Tolerance

1

Fault Tolerance

- Terminology & Background
- Byzantine Fault Tolerance
- Issues in client/server
- Reliable group communication

2

Fault Tolerance

- Being fault tolerant is strongly related to what are called dependable systems. Dependability implies the following:
  - **Availability**: probability the system operates correctly at any given moment
  - **Reliability**: ability to run correctly for a long interval of time
  - **Safety**: failure to operate correctly does not lead to catastrophic failures
  - **Maintainability**: ability to "easily" repair a failed system

3

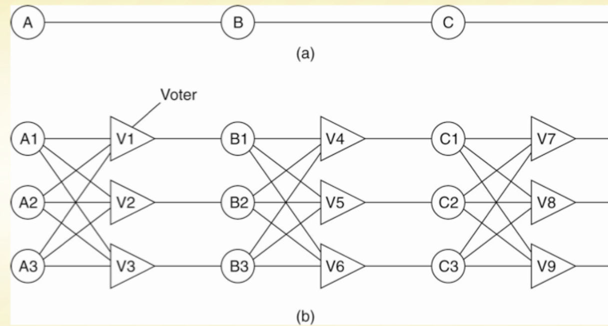
Failure Models

Type of failure	Description
Crash failure	A server halts, but is working correctly until it halts
Omission failure *Receive omission* *Send omission*	A server fails to respond to incoming requests A server fails to receive incoming messages A server fails to send messages
Timing failure	A server's response lies outside the specified time interval
Response failure *Value failure* *State transition failure*	A server's response is incorrect The value of the response is wrong The server deviates from the correct flow of control
Arbitrary failure	A server may produce arbitrary responses at arbitrary times

- A system is said to fail if it cannot meet its promises. An error on the part of a system's state may lead to a failure. The cause of an error is called a fault.

4

Failure Masking by Redundancy



- Triple modular redundancy. For each voter, if two or three of the inputs are the same, the output is equal to the input. If all three inputs are different, the output is undefined.

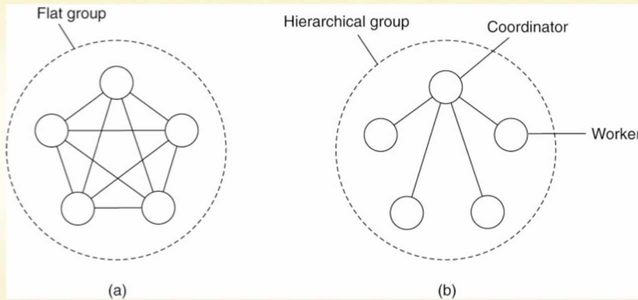
5

Process Resilience - 1

- The key approach to tolerating a faulty process is to use process groups:
  - This group can be thought of as an abstraction for a single "process". Messages to the "process" are sent to the entire group.
  - Group membership can be dynamic
    - Need mechanisms for creating and destroying groups
    - Need mechanisms for adding and removing processes from groups
  - Many choices for the structure of the group

6

Flat Groups versus Hierarchical Groups



- (a) Communication in a flat group.
- (b) Communication in a simple hierarchical group.

7

Process Resilience - 2

- Reaching agreement:
  - computation results
  - Electing a leader
  - synchronization
  - committing to a transaction
  - …
- How much replication is necessary?
  - A system is k fault tolerant if it can survive faults in k components and still meet its specifications.

8

Agreement in Faulty Systems - 1

- Many things can go wrong…
- Communication
  - Message transmission can be unreliable
  - Time taken to deliver a message is unbounded
  - Adversary can intercept messages
- Processes
  - Can fail or team up to produce wrong results
- Agreement very hard, sometime impossible, to achieve!

9

Agreement in Faulty Systems - 2

- Possible characteristics of the underlying system:

1. Synchronous versus asynchronous systems.
   - A system is synchronized if the process operation in lock-step mode. Otherwise, it is asynchronous.
2. Communication delay is bounded or not.
3. Message delivery is ordered or not.
4. Message transmission is done through unicasting or multicasting.

10

Agreement in Faulty Systems - 3

Process behavior	Message ordering: Unordered, Unicast	Unordered, Multicast	Ordered, Unicast	Ordered, Multicast	Communication delay
Synchronous			X		Bounded
Synchronous			X		Unbounded
Asynchronous	X	X	X	X	Bounded
Asynchronous			X	X	Unbounded

Message transmission

- Circumstances under which distributed agreement can be reached. Note that most distributed systems assume that
  1. processes behave asynchronously
  2. messages are unicast
  3. communication delays are unbounded (see red blocks)

11

Fault Tolerance

- Terminology & Background
- Byzantine Fault Tolerance
- Issues in client/server
- Reliable group communication

12

Agreement in Faulty Systems - 4

- Byzantine Agreement [Lamport, Shostak, Pease, 1982]
- Assumptions:
  - Every message that is sent is delivered correctly
  - The receiver knows who sent the message
  - Message delivery time is bounded

Message ordering / Message transmission / Process behavior / Communication delay

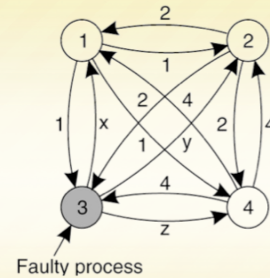
Process behavior	Unordered: Unicast	Unordered: Multicast	Ordered: Unicast	Ordered: Multicast	Communication delay
Synchronous			X		Bounded
Synchronous			X		Unbounded
Asynchronous	X	X	X	X	Bounded
Asynchronous			X	X	Unbounded

13

Agreement in Faulty Systems - 5

System of N processes, where each process i will provide a value v_i to each other. Some number of these processes may be incorrect (or malicious)

Goal: Each process learn the true values sent by each of the correct processes



- Figure 8-5. The Byzantine agreement problem for three nonfaulty and one faulty process.

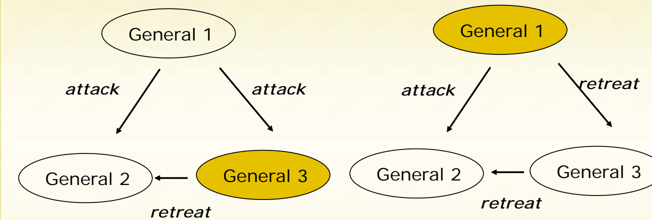
14

Byzantine General's Problem

- The Problem: "Several divisions of the Byzantine army are camped outside an enemy city, each division commanded by its own general. After observing the enemy, they must decide upon a common plan of action. Some of the generals may be traitors, trying to prevent the loyal generals from reaching agreement."
- Goal:
  - All loyal generals decide upon the same plan of action.
  - A small number of traitors cannot cause the loyal generals to adopt a bad plan.
- The paper considers a slightly different version from the standpoint of one general (i.e. process) and multiple lieutenants.
- Goal:
  - All loyal lieutenants obey the same order.
  - If the commanding general is loyal, the every loyal lieutenant obeys the order he sends.

Lamport, Shostak, Pease. The Byzantine General's Problem. ACM TOPLAS, 4,3, July 1982, 382-401. 15

Impossibility Results



- *No solution for three processes can handle a single traitor.*
- *In a system with m faulty processes agreement can be achieved only if there are 2m+1 (more than 2/3) functioning correctly.*

Lamport, Shostak, Pease. The Byzantine General's Problem. ACM TOPLAS, 4,3, July 1982, 382-401. 16

Byzantine Agreement Algorithm (oral messages) - 1

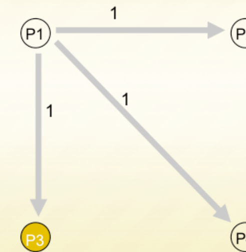
- Phase 1: Each process sends its value to the other processes. Correct processes send the same (correct) value to all. Faulty processes may send different values to each if desired (or no message).

- *Assumptions: 1) Every message that is sent is delivered correctly; 2) The receiver of a message knows who sent it; 3) The absence of a message can be detected.*

Lamport, Shostak, Pease. The Byzantine General's Problem. ACM TOPLAS, 4,3, July 1982, 382-401. 17

Byzantine General Problem Example - 1

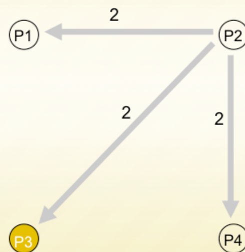
- Phase 1: Generals announce their troop strengths to each other



18

Byzantine General Problem Example - 2

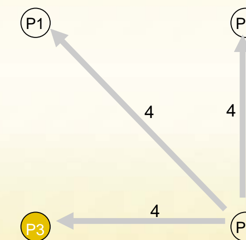
- Phase 1: Generals announce their troop strengths to each other



19

Byzantine General Problem Example - 3

- Phase 1: Generals announce their troop strengths to each other



20

Byzantine Agreement Algorithm (oral messages) - 2

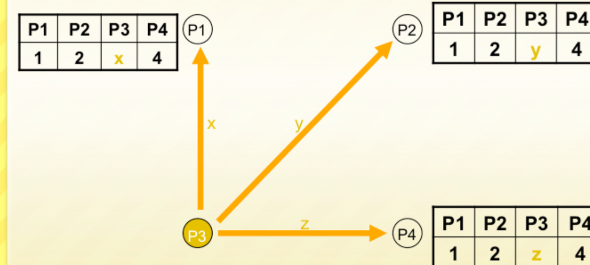
- Phase 2: Each process uses the messages to create a vector of responses – must be a default value for missing messages.

- *Assumptions: 1) Every message that is sent is delivered correctly; 2) The receiver of a message knows who sent it; 3) The absence of a message can be detected.*

Lamport, Shostak, Pease. The Byzantine General's Problem. ACM TOPLAS, 4,3, July 1982, 382-401. 21

Byzantine General Problem Example - 4

- Phase 2: Each general construct a vector with all troops



22

Byzantine Agreement Algorithm (oral messages) - 3

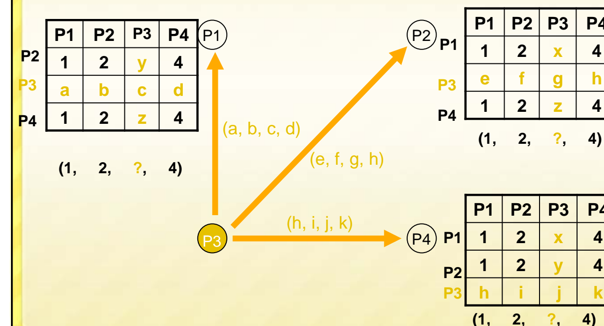
- Phase 3: Each process sends its vector to all other processes.
- Phase 4: Each process the information received from every other process to do its computation.

- *Assumptions: 1) Every message that is sent is delivered correctly; 2) The receiver of a message knows who sent it; 3) The absence of a message can be detected.*

Lamport, Shostak, Pease. The Byzantine General's Problem. ACM TOPLAS, 4,3, July 1982, 382-401. 23

Byzantine General Problem Example - 5

- Phase 3,4: Generals send their vectors to each other and compute majority voting



24

Byzantine Agreement Algorithm (oral messages) - 4

- Byzantine Agreement:
- Note: This result only guarantees that each process receives the true values sent by correct processors, but it does not identify the correct processes!

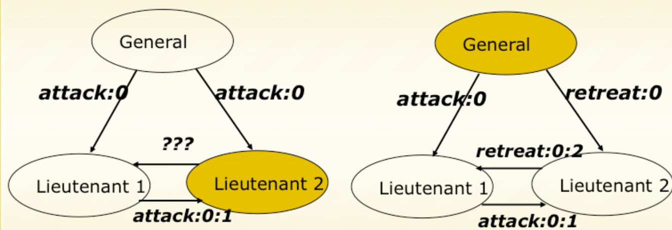
Lamport, Shostak, Pease. The Byzantine General's Problem. ACM TOPLAS, 4,3, July 1982, 382-401. 25

Byzantine Agreement Algorithm (signed messages)

- Adds the additional assumptions:
  - A loyal general's signature cannot be forged and any alteration of the contents of the signed message can be detected.
  - Anyone can verify the authenticity of a general's signature.
- Algorithm SM(m):
- The general signs and sends his value to every lieutenant.
- For each i:
  - If lieutenant i receives a message of the form v:0 from the commander and he has not received any order, then he lets Vi equal {v} and he sends v:0:i to every other lieutenant.
  - If lieutenant i receives a message of the form v:0:j1:…:jk and v is not in the set Vi then he adds v to Vi and if k < m, he sends the message v:0:j1:…:jk:i to every other lieutenant other than j1,…,jk
  - For each i: When lieutenant i will receive no more messages, he obeys the order in choice(Vi).
- Algorithm SM(m) solves the Byzantine General's problem if there are at most m traitors.

Lamport, Shostak, Pease. The Byzantine General's Problem. ACM TOPLAS, 4,3, July 1982, 382-401. 26

Signed messages



SM(1) with one traitor

Lamport, Shostak, Pease. The Byzantine General's Problem. ACM TOPLAS, 4,3, July 1982, 382-401. 27

Fault Tolerance

- Terminology & Background
- Byzantine Fault Tolerance
- Issues in client/server
- Reliable group communication

28

Fault Tolerance in Client/Server Systems

- Five different classes of failures that can occur in RPC systems:

1. The client is unable to locate the server. Can be dealt with at the client.
2. The request message from the client to the server is lost.
3. The server crashes after receiving a request.
4. The reply message from the server to the client is lost.
5. The client crashes after sending a request.

29

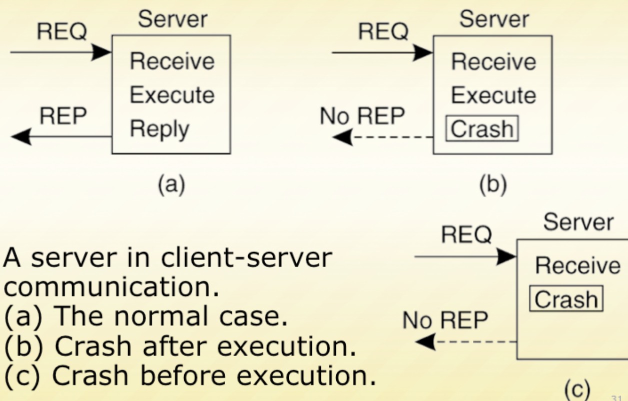
Lost Messages

2. The request message from the client to the server is lost.
5. The reply message from the server to the client is lost.

- Timers at OS level can be used to detect lost messages.
- From the client standpoint – these two cases look the same but they aren't.
- Idempotent messages aren't a problem.
- Client can safely re-issue a message that isn't idempotent if there is some way (sequence numbers, stamps) for a server to detect the re-issue
  - Basically make them idempotent

30

Server Crashes (1)



- A server in client-server communication.
(a) The normal case.
(b) Crash after execution.
(c) Crash before execution.

31

Server Crashes (2)

- No way for client to differentiate between the two crash cases (b) and (c).
- How should client react? There several options:
  - At-least-once semantics– client keeps trying (sending messages) until a reply is received.
  - At-most-once semantics – client gives up
  - No guarantees

32

Server Crashes (3)

- Consider scenario where a client sends text to a print server.
- There are three events that can happen at the server:
  - Send the completion message (M),
  - Print the text (P),
  - Crash (C) – at recovery, send 'recovery' message to clients.
- Server strategies:
  - send completion message before printing
  - send completion message after printing

33

Server Crashes (4)

- These events can occur in six different orderings:

1. M →P →C: A crash occurs after sending the completion message and printing the text.
2. M →C (→P): A crash happens after sending the completion message, but before the text could be printed.
3. P →M →C: A crash occurs after sending the completion message and printing the text.
4. P→C(→M): The text printed, after which a crash occurs before the completion message could be sent.
5. C (→P →M): A crash happens before the server could do anything.
6. C (→M →P): A crash happens before the server could do anything.

34

Server Crashes (5)

- Client strategies after a crash:
  - do nothing (i.e. do not re-issue request)
  - Always re-issue request
  - Re-issue only if request acknowledged
  - Re-issue only if request not acknowledged.

35

Server Crashes (6)

- Different combinations of client and server strategies in the presence of server crashes.

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 = Text is printed twice
ZERO = Text is not printed at all

36

Client Crashes

- Can create orphans (unwanted computations) that waste CPU, potentially lock up resources and create confusion when client re-boots.
- Nelson solutions:
  1. Orphan Extermination – keep a log of RPCs at client that is checked at re-boot time to remove orphans.
  2. Reincarnation – divide time into epochs. After a client re-boot, increment its epoch and kill off any of its requests belonging to an earlier epoch.
  3. Gentle Reincarnation – at reboot time, an epoch announcement causes all machines to locate the owners of any remote computations.
  4. Expiration – each RPC is given time T to complete (but a live client can ask for more time)

Nelson. Remote Procedure Call. Ph.D. Thesis, CMU, 1981.

37

Fault Tolerance

- Terminology & Background
- Byzantine Fault Tolerance
- Issues in client/server
- Reliable group communication

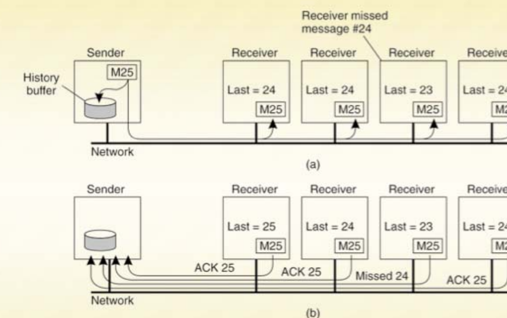
38

Reliable Group Communication

- Can we guarantee that all members of a process group receive all messages delivered to that group?
- Simplest solutions assume that we have a small number of processes in the group, processes do not fail, and the group does not change during message transmission.
- Approaches that rely on feedback (acknowledgements) do not scale well
  - Will look at this later in semester

39

Basic Reliable-Multicasting Schemes



- A simple solution to reliable multicasting when all receivers are known and are assumed not to fail
  - (a) Message transmission. (b) Reporting feedback.

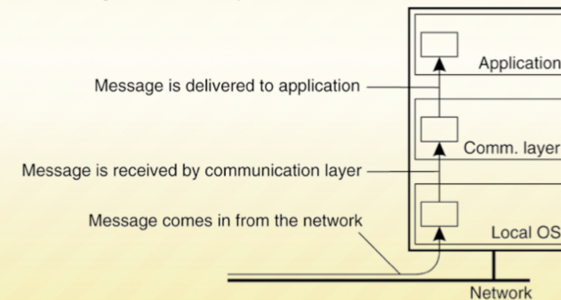
40

Atomic Multicast

- All messages are delivered in the same order to "all" processes
- **Group view**: the set of processes known by the sender when it multicast the message
- **Virtual synchronous multicast**: a message multicast to a group view G is delivered to all nonfaulty processes in G
  - If sender fails after sending the message, the message may be delivered to no one

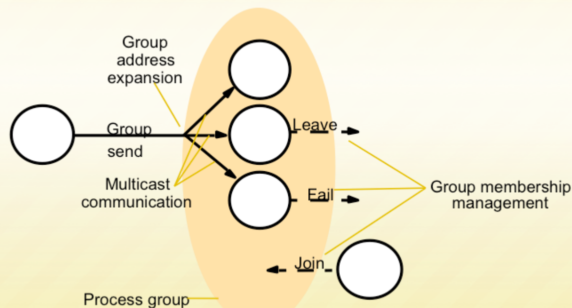
Virtual Synchrony (1)

- Logical organization of a distributed system to distinguish between message receipt and message delivery.



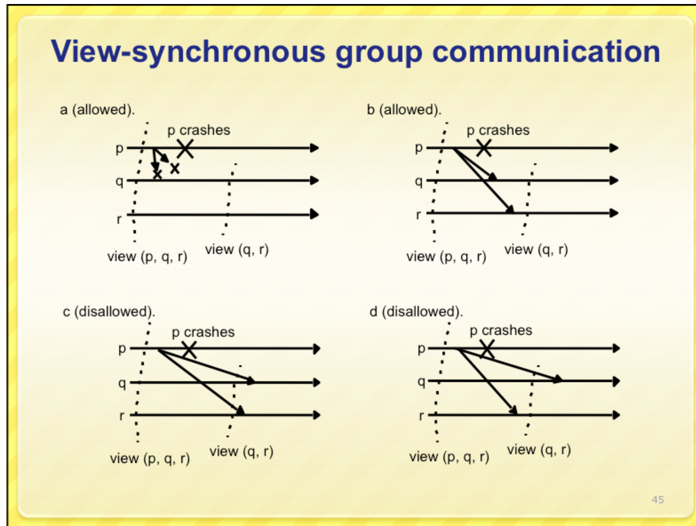
Group communication

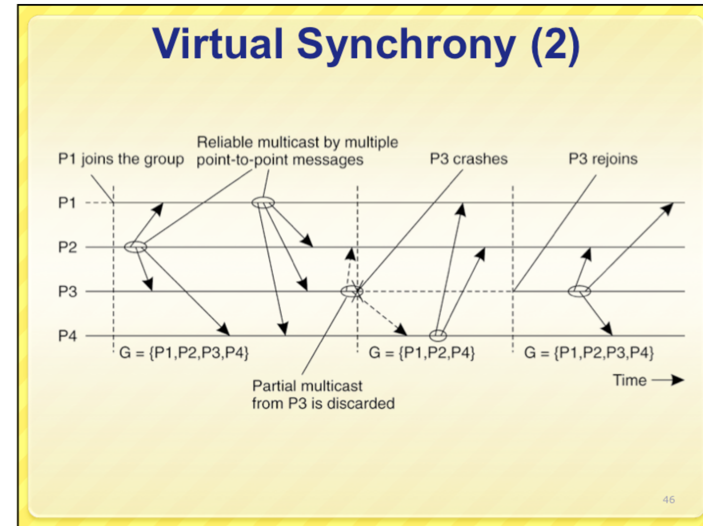
- Group membership service
  - Provides an interface for group membership changes
  - Implements a failure detector
  - Notifies members of group membership changes



View delivery

- A view reflects current membership of group
- A view is delivered when a membership change occurs and the application is notified of the change
- View-synchronous group communication
  - the delivery of a new view draws a conceptual line across the system and every message is either delivered on side or the other of that line





Virtual Synchrony Implementation: [Birman et al., 1991]

- Only stable messages are delivered
- **Stable message**: a message received by all processes in the message's group view
- Assumptions (can be ensured by using TCP):
  - Point-to-point communication is reliable
  - Point-to-point communication ensures FIFO-ordering

47

Message Ordering (1)

- Four different orderings are distinguished:
  1. Unordered multicasts
  2. FIFO-ordered multicasts
  3. Causally-ordered multicasts
  4. Totally-ordered multicasts
- Atomicity (everyone sees same order) is an orthogonal property

48

Unordered Multicast

Process P1	Process P2	Process P3
sends m1	receives m1	receives m2
sends m2	receives m2	receives m1

- Figure 8-14. Three communicating processes in the same group. The ordering of events per process is shown along the vertical axis.

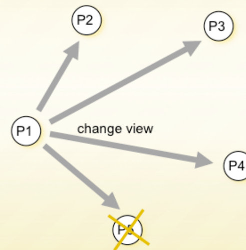
FIFO Multicast

Process P1	Process P2	Process P3	Process P4
sends m1	receives m1	receives m3	sends m3
sends m2	receives m3	receives m1	sends m4
	receives m2	receives m2	
	receives m4	receives m4	

- Figure 8-15. Four processes in the same group with two different senders, and a possible delivery order of messages under FIFO-ordered multicasting

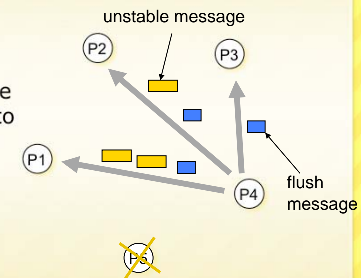
Virtual Synchrony Implementation: Example

- G_i = {P1, P2, P3, P4, P5}
- P5 fails
- P1 detects that P5 has failed
- P1 send a "view change" message to every process in G_{i+1} = {P1, P2, P3, P4}



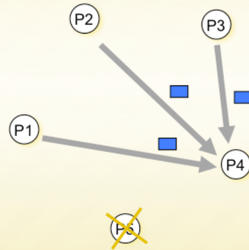
Virtual Synchrony Implementation: Example

- Every process
  - Send each unstable message m from Gi to members in G_{i+1}
  - Marks m as being stable
  - Send a flush message to mark that all unstable messages have been sent



Virtual Synchrony Implementation: Example

- Every process
  - After receiving a flush message from any process in G_{i+1} installs G_{i+1}



53

Important Lessons

- Terminology & Background
  - Failure models
- Byzantine Fault Tolerance
  - Protocol design → with and without crypto
  - How many servers do we need to tolerate
- Issues in client/server
  - Where do all those RPC failure semantics come from?
- Reliable group communication
  - How do we manage group membership changes as part of reliable multicast

54