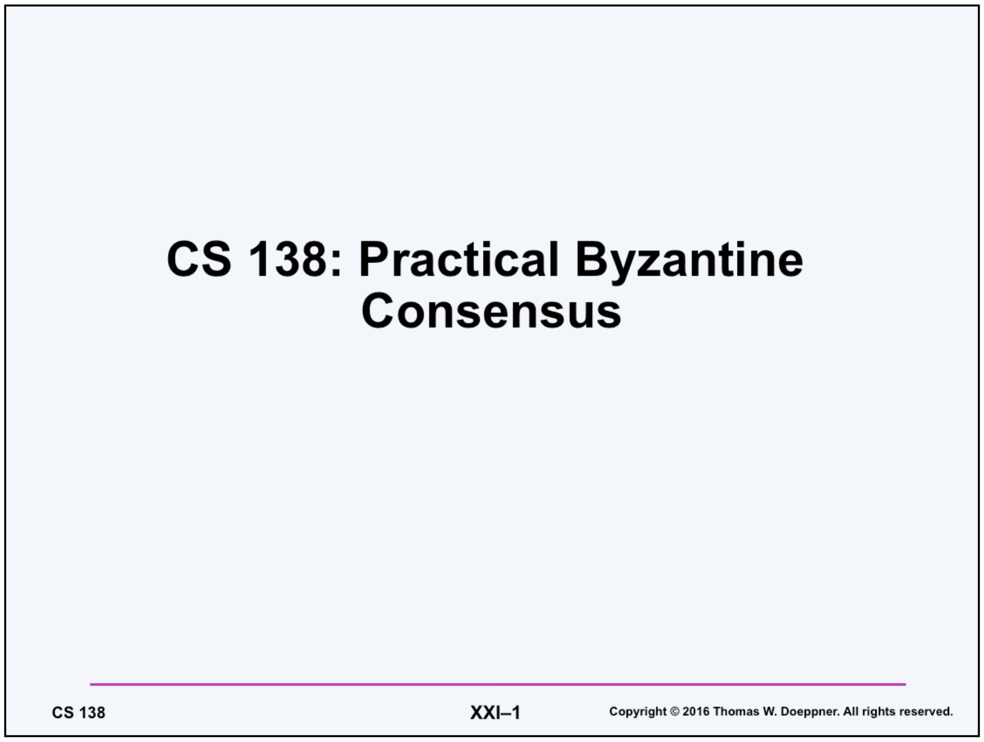
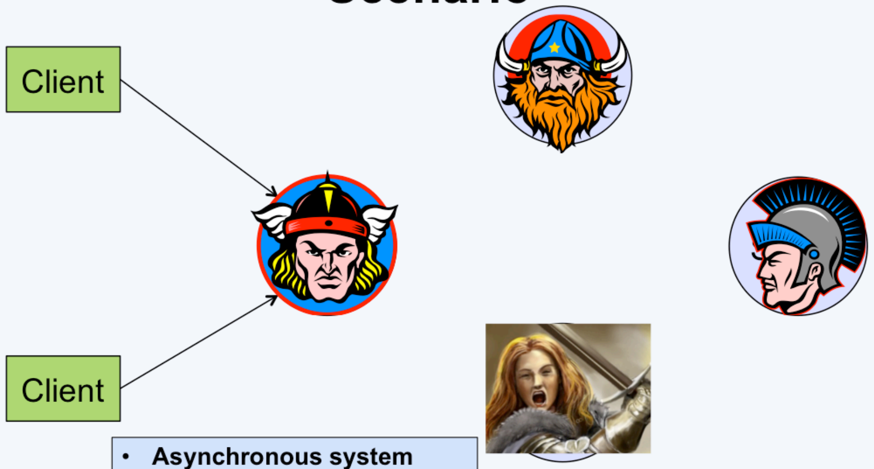


CS 138: Practical Byzantine Consensus

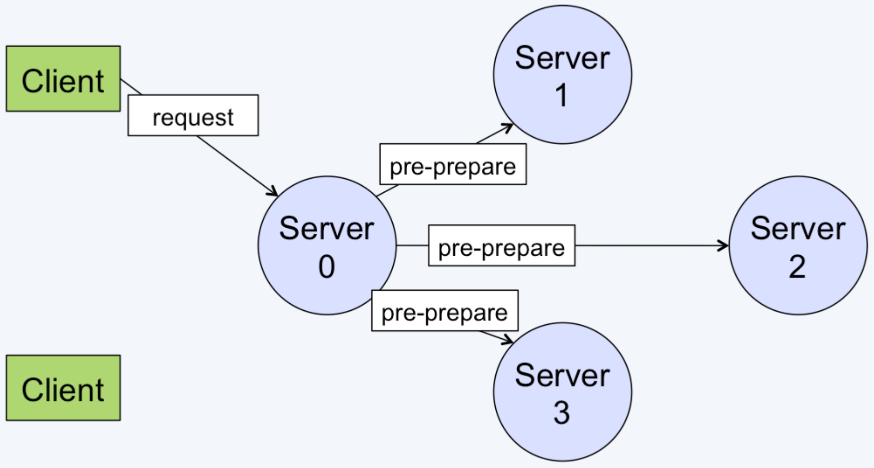
This lecture is based on "Practical Byzantine Fault Tolerance," by M. Castro and B. Liskov, published in the Proceedings of the Third Symposium on Operating Systems Design and Implementation, New Orleans, USA, February 1999. It can be found at http://research.microsoft.com/en-us/um/people/mcastro/publications/osdi99.pdf. The approach described here is also used for the inner ring in OceanStore.

Scenario

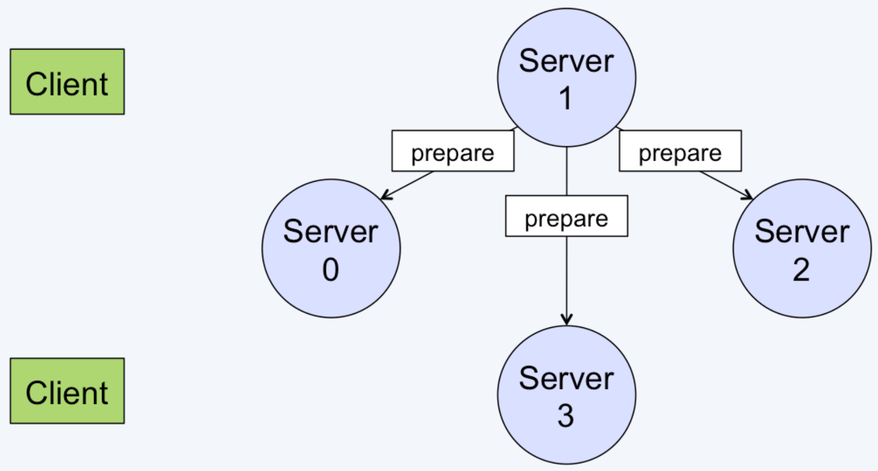


- Asynchronous system
- Signed messages
- Servers are state machines
- It has to be practical

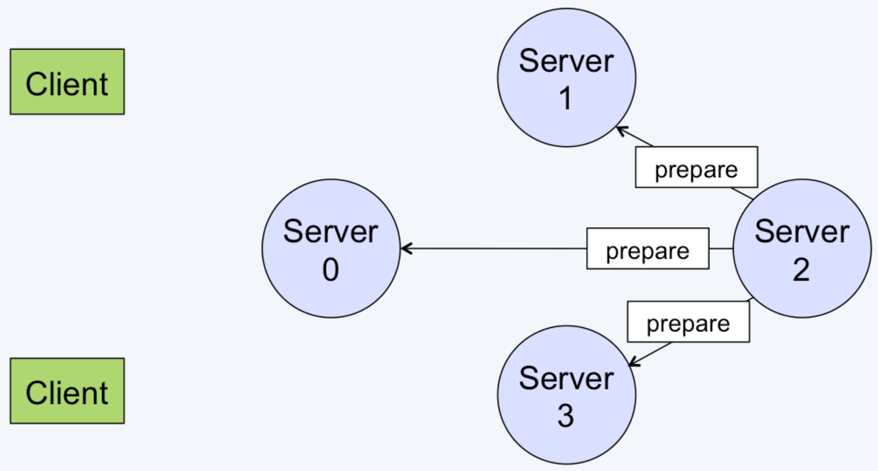
The Request


Client
request
Server 0
pre-prepare
Server 1
pre-prepare
Server 2
pre-prepare
Server 3
Client

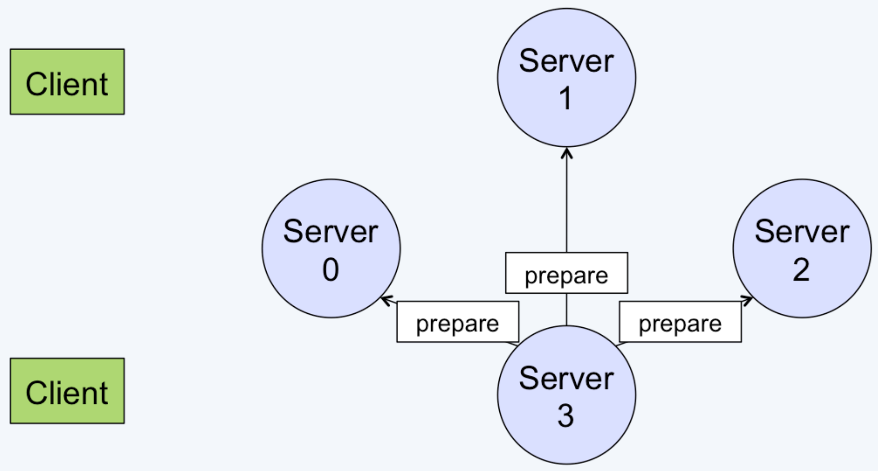
Non-Primaries Respond (1)

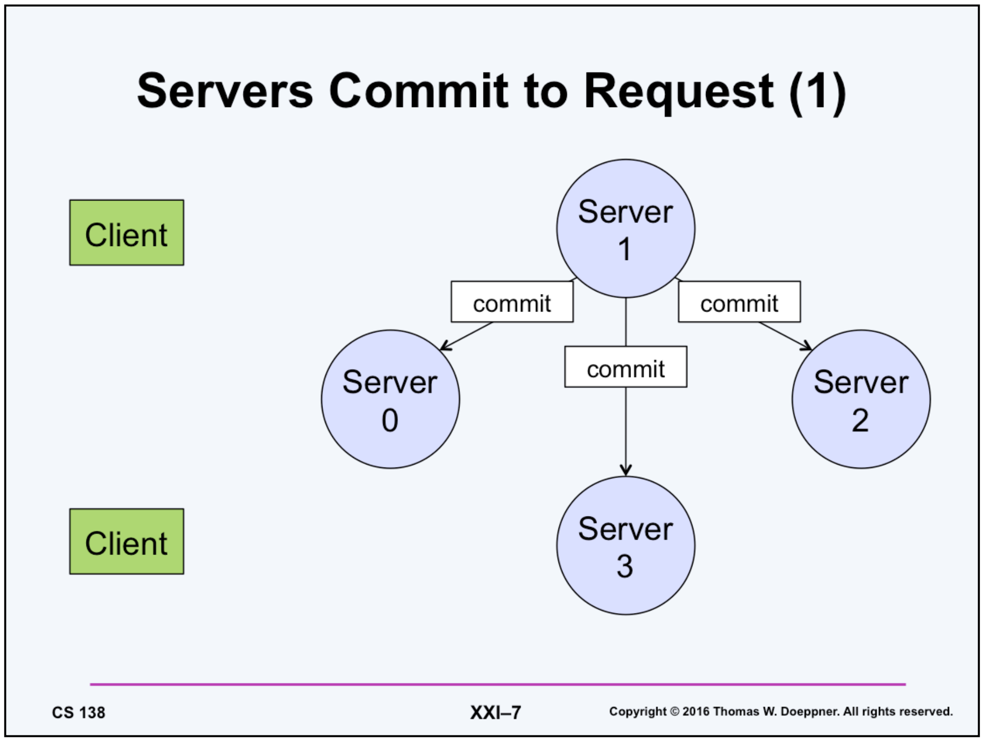


Non-Primaries Respond (2)

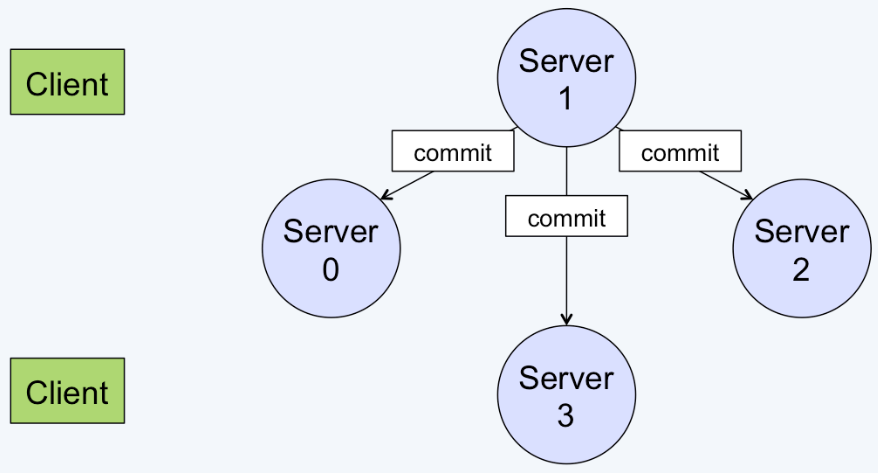


Non-Primaries Respond (3)

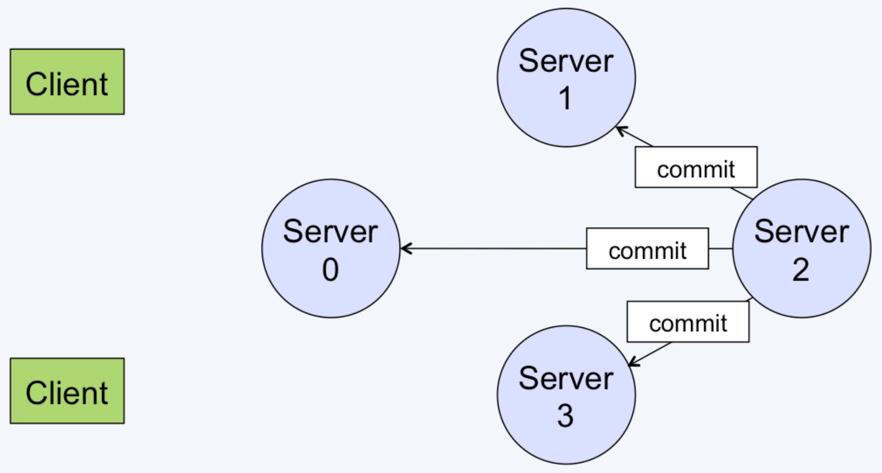




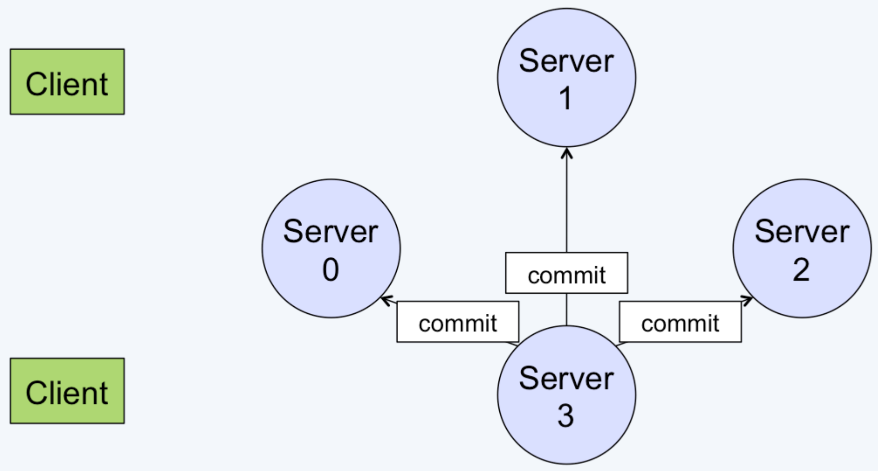
Servers Commit to Request (1)



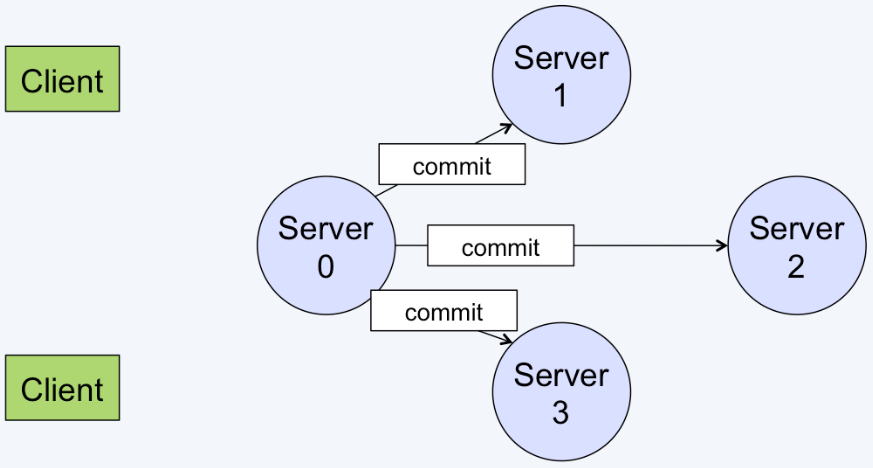
Servers Commit to Request (2)

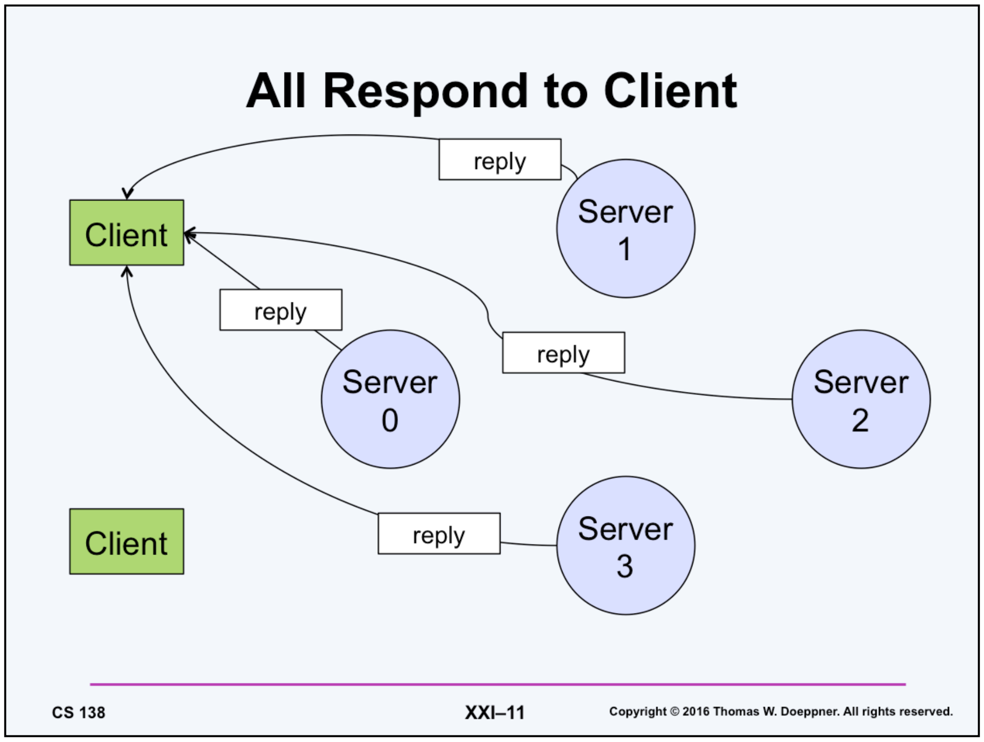


Servers Commit to Request (3)



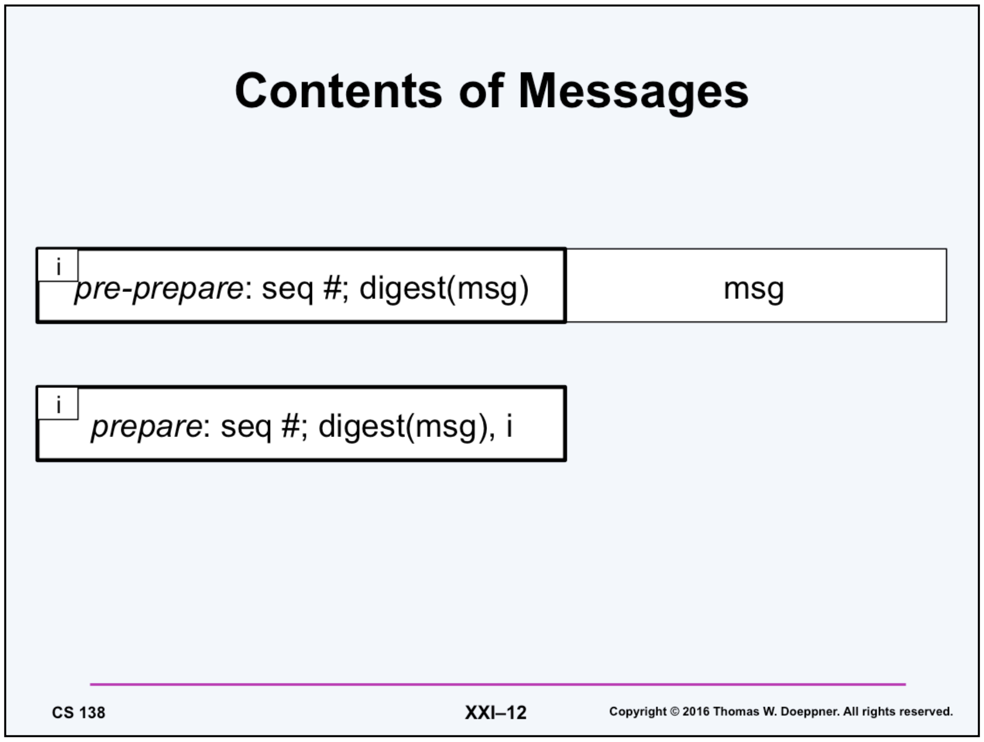
Servers Commit to Request (4)





All Respond to Client

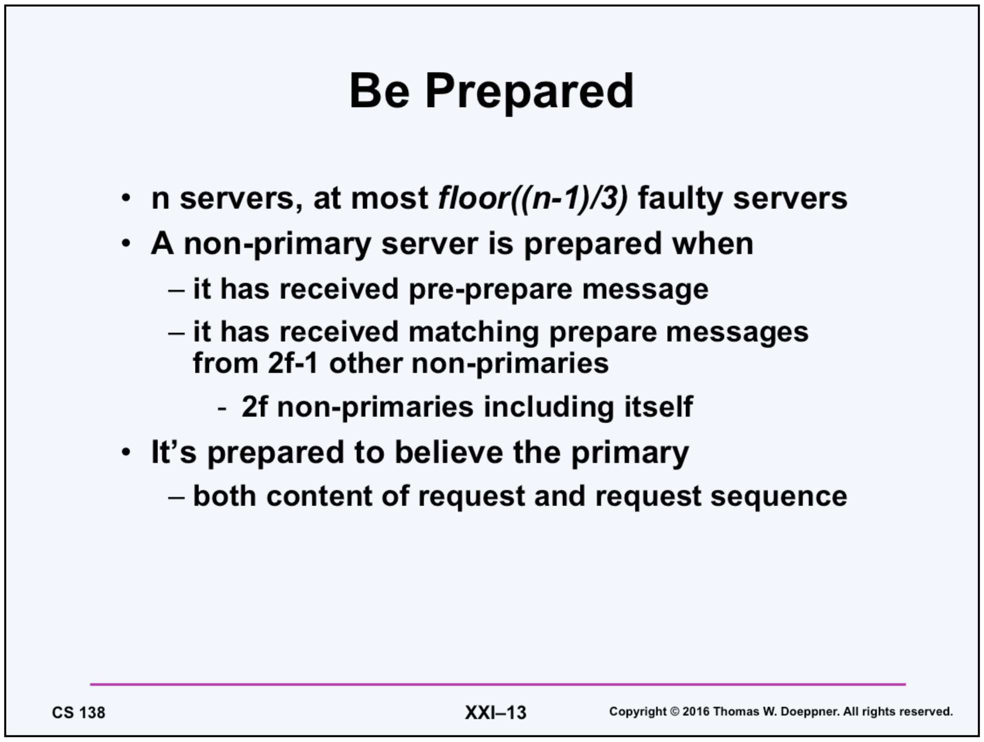
The client believes the result once it has received *f+1* identical replies, where at most *f* servers are faulty.



Contents of Messages

i
pre-prepare: seq #; digest(msg)
msg
i
prepare: seq #; digest(msg), i

The thick outline with the box containing *i* in the upper left-hand corner means the contents of the larger box are signed by server *i*.


Be Prepared

- n servers, at most *floor((n-1)/3)* faulty servers
- A non-primary server is prepared when
  - it has received pre-prepare message
  - it has received matching prepare messages from 2f-1 other non-primaries
    - 2f non-primaries including itself
- It's prepared to believe the primary
  - both content of request and request sequence

However …

- There are multiple clients, each sending a sequence of requests
- Communication isn't perfect
  - – messages may arrive out of order
- Server s may be prepared, but s′ is not
  - – but will be eventually
- Server may be prepared for request q but not for q-1

Commitment

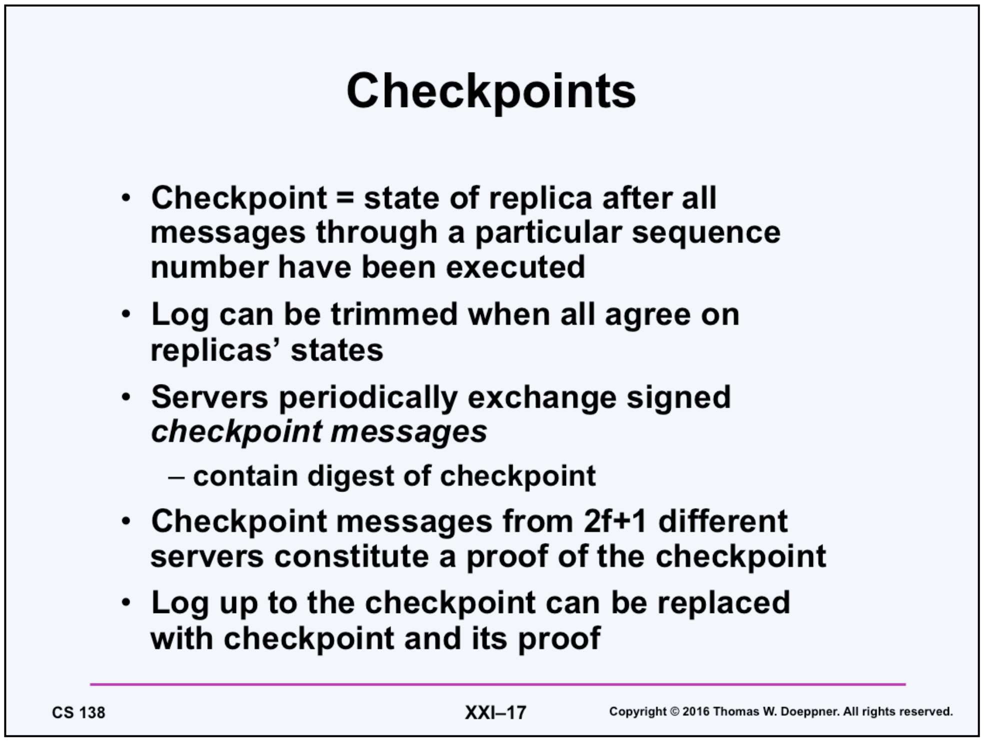
- Server i multicasts *commit* message to all others when it is prepared

  i | *commit*: seq #; digest(msg), i

- A message is *committed* if it is prepared at f+1 non-faulty servers
  - how does an individual server know this?
    - it is prepared and has received 2f commits from others
- Server executes message when
  - message is committed
  - and all previous messages have been executed

Logging

- Each server maintains log of
  - pre-prepares
  - prepares
  - commits



Checkpoints

- Checkpoint = state of replica after all messages through a particular sequence number have been executed
- Log can be trimmed when all agree on replicas' states
- Servers periodically exchange signed *checkpoint messages*
  - contain digest of checkpoint
- Checkpoint messages from 2f+1 different servers constitute a proof of the checkpoint
- Log up to the checkpoint can be replaced with checkpoint and its proof

Traitorous Primary

- Client sends request
- No response from primary
- Client re-sends request to all servers
- Servers forward request to primary
- If no response, then need new primary

Views

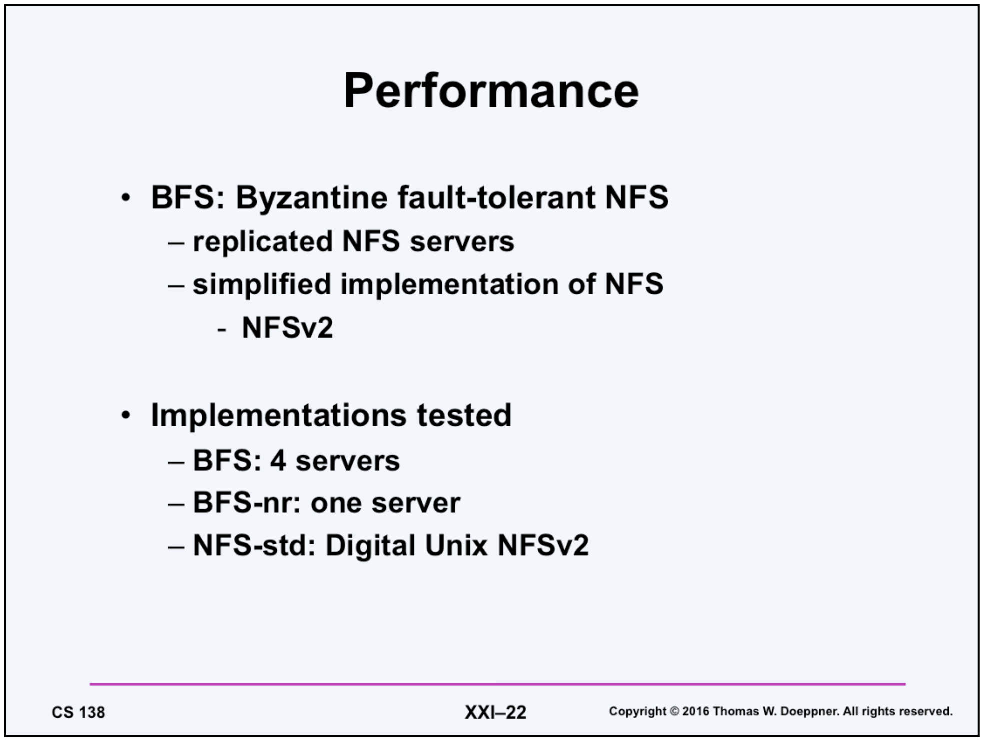
- A particular primary server is in charge of a view *v*
- If the primary changes, the view changes to *v +1*
  - the primary for view v is server v mod S
    - S is the number of servers

View Changes (1)

- Non-primaries who time-out waiting for server send signed *view-change* messages
  - provide
    - most recent checkpoint plus proof
    - list of prepared messages since checkpoint
      - with proof: pre-prepare plus prepare messages

View Changes (2)

- New primary, after receiving 2f valid *view-change* messages, responds with *new-view* message
  - provides
    - set of view-change messages
      - i.e., proof of view change
    - list of pre-prepare messages for all prepared messages since checkpoint
      - missing messages are nullified
  - non-primaries move to new view and reprocess prepared messages in this view



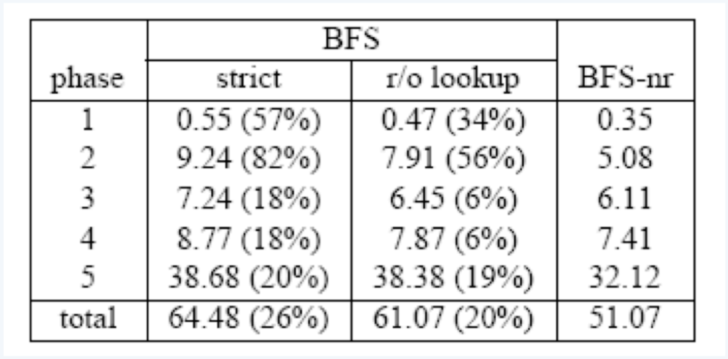
Performance

- BFS: Byzantine fault-tolerant NFS
  - replicated NFS servers
  - simplified implementation of NFS
    - NFSv2
- Implementations tested
  - BFS: 4 servers
  - BFS-nr: one server
  - NFS-std: Digital Unix NFSv2

Andrew Benchmark

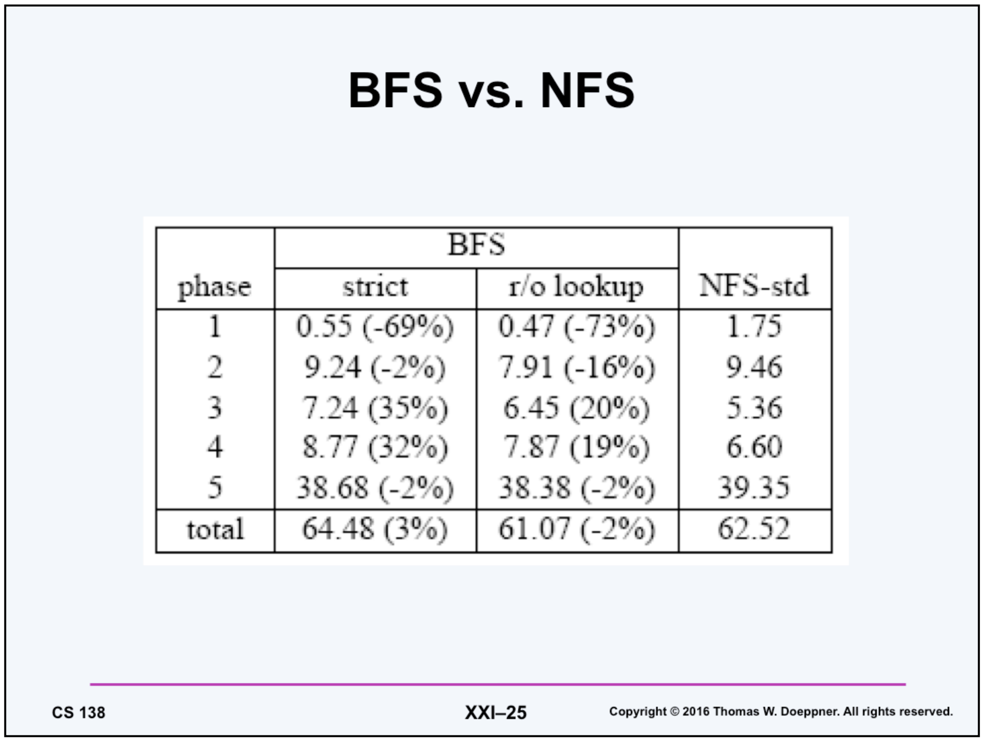
- phase 1
  - – creates subdirectories recursively
- phase 2
  - – copies a source-code tree
- phase 3
  - – examines status of all files without reading their data
- phase 4
  - – reads all data bytes
- phase 5
  - – compiles and links all files

BFS vs BFS-nr



phase	BFS strict	BFS r/o lookup	BFS-nr
1	0.55 (57%)	0.47 (34%)	0.35
2	9.24 (82%)	7.91 (56%)	5.08
3	7.24 (18%)	6.45 (6%)	6.11
4	8.77 (18%)	7.87 (6%)	7.41
5	38.68 (20%)	38.38 (19%)	32.12
total	64.48 (26%)	61.07 (20%)	51.07

This table is taken from the aforementioned paper. The “r/o lookup” column is the result of modifying BFS so that reading data does not cause the “time of last access” to be modified, making it truly read-only. The times are in seconds. The percentages indicate how much slower things were than when done with BFS-nr (no replication).



BFS vs. NFS

phase	BFS		NFS-std
	strict	r/o lookup	
1	0.55 (-69%)	0.47 (-73%)	1.75
2	9.24 (-2%)	7.91 (-16%)	9.46
3	7.24 (35%)	6.45 (20%)	5.36
4	8.77 (32%)	7.87 (19%)	6.60
5	38.68 (-2%)	38.38 (-2%)	39.35
total	64.48 (3%)	61.07 (-2%)	62.52

This looks impressive. However, NFS-std is NFS v2, which has synchronous writes. BFS does not have synchronous writes, on the theory that they are unneeded given the replication. A better comparison would have been with NFSv3. Also, BFS does not have a real local file system, but a simplified file system that is not has robust as that used by NFS and, most likely, much faster as a result.