Replication and Recovery in Distributed Systems

ECE 677

University of Arizona

Replication

Advantages :

- reduce communication traffic, => improve response time
- increase system availability, => reduce the effect of server and communication failures
- several clients requests can be handled in parallel => improve system throughput

Replication Management Techniques

- ❑ Active Redundancy
- ❑ Passive Redundancy
- ❑ Semi Active Redundancy

Replication Abstract Model

Request (RE): the client submits an operation to one (or more) replicas.

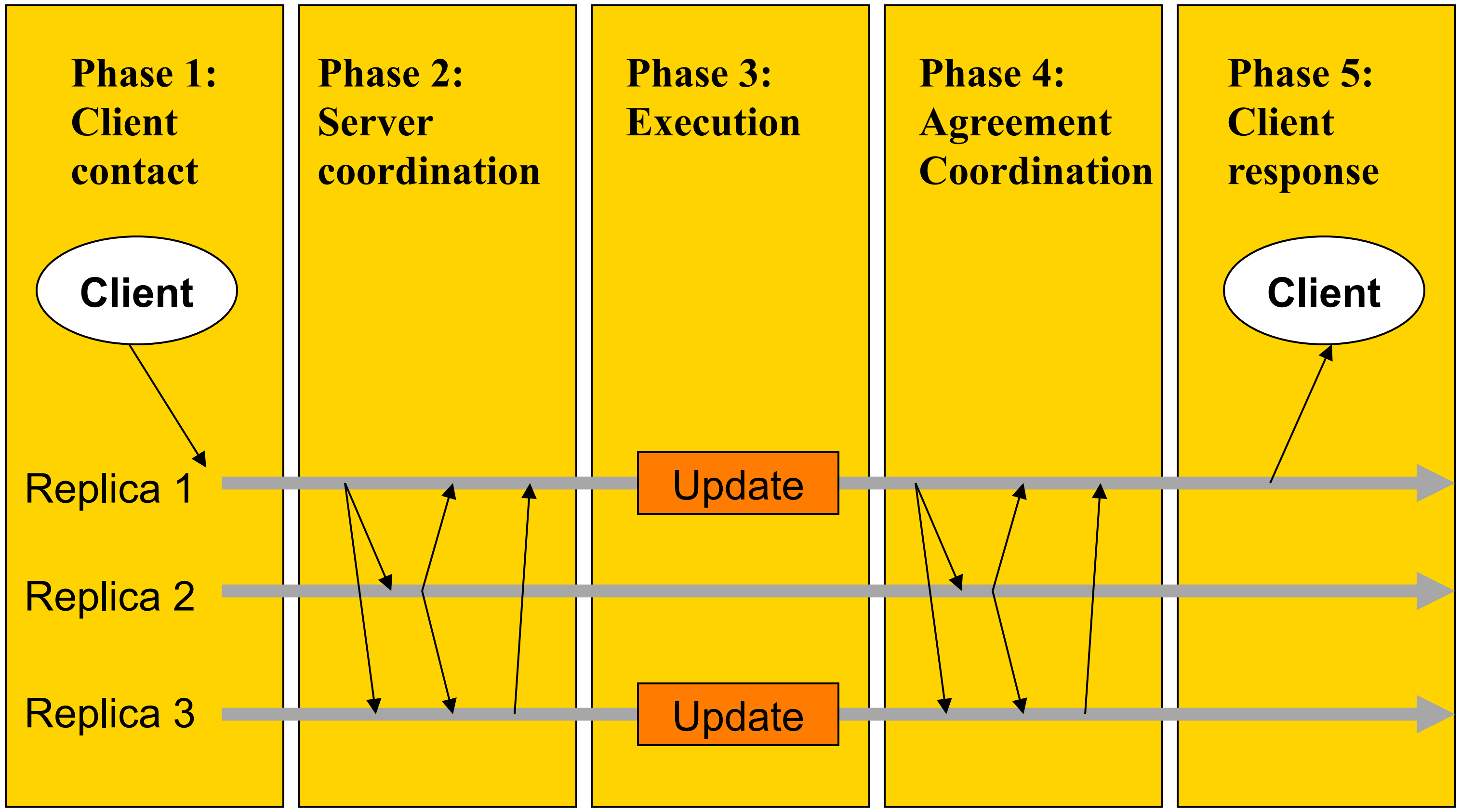
Server coordination (SC): the replica servers coordinate with each other to synchronize the execution of the operation (ordering of concurrent operations).

Execution (EX): the operation is executed on the replica servers.

Agreement coordination (AC): the replica servers agree on the result of the execution (e.g., to guarantee atomicity).

Response (END): the outcome of the operation is transmitted back to the client.

Abstract model with the five phases



Replication in Distributed System

Three replication techniques.

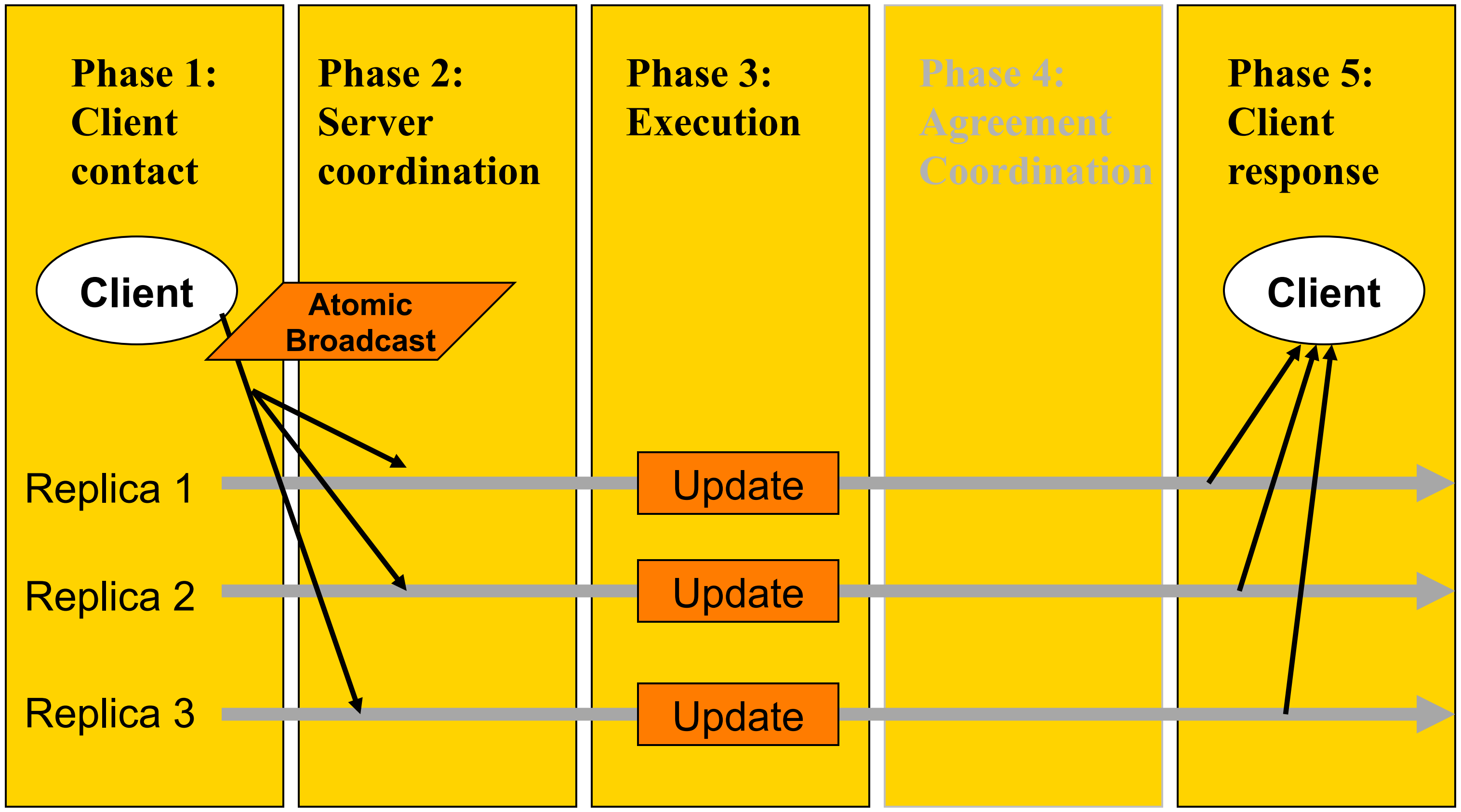
1. *Active Replication,*
2. *2. Passive Replication,*
3. *3. Semi-Active Replication.*

Replication in Distributed System

Active Replication (Modular Redundancy)

1. The client sends the request to the servers using an Atomic Broadcast.
2. Server coordination is given by the total order property of the Atomic Broadcast.
3. All replicas execute the request in the order they are delivered.
4. No coordination is necessary, as all replica process the same request in the same order. Because replica are deterministic, they all produce the same results.
5. All replica send back their result to the client, and the client typically only waits for the first answer.

Active Replication

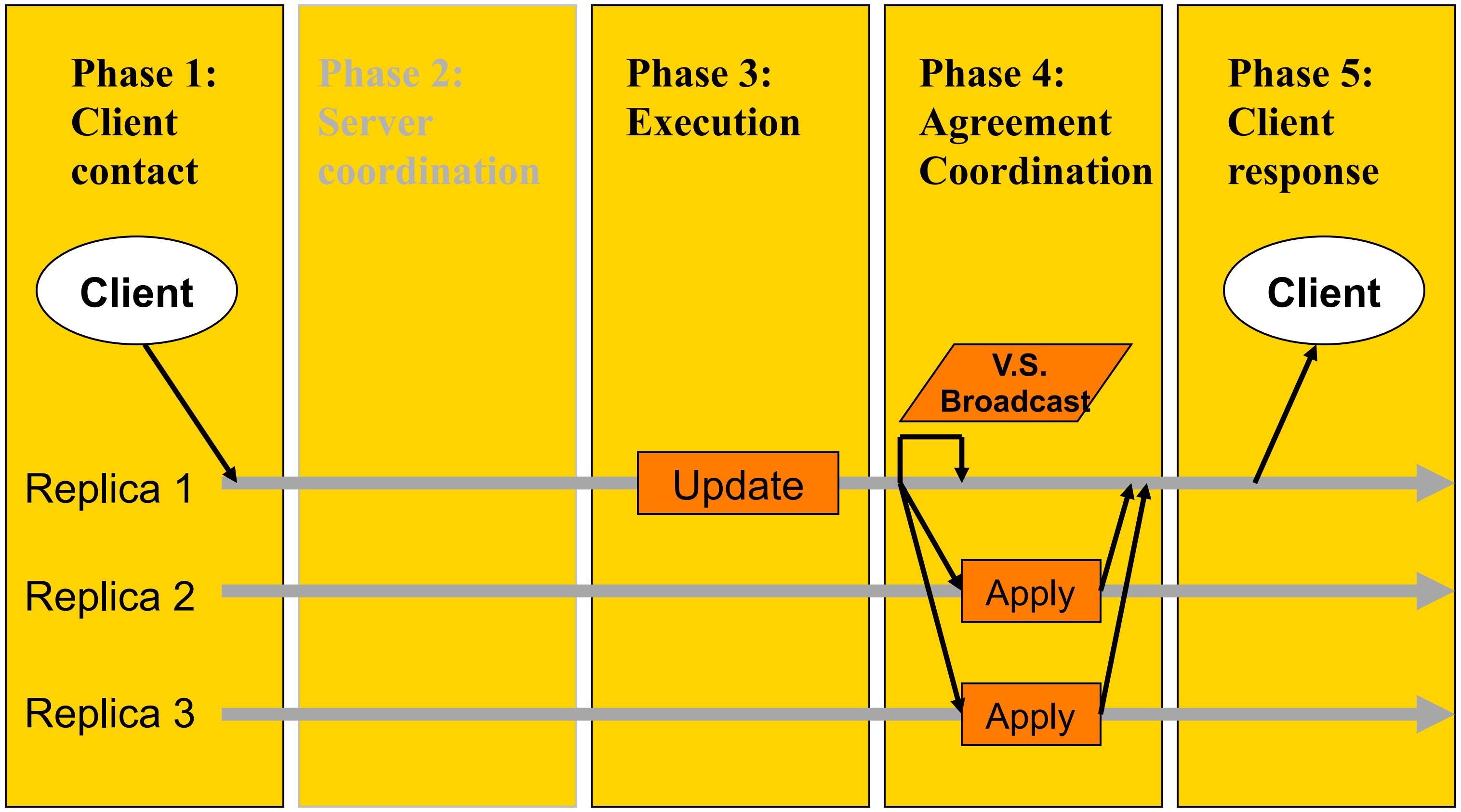


Replication in Distributed System

Passive Replication (Primary-stand-by)

1. The client sends the request to the primary.
2. No initial coordination is needed.
3. The request is executed in the primary.
4. By sending the updated information to the backups, the primary coordinates with the other replicas.
5. The primary sends the answer to the client.

Passive Replication

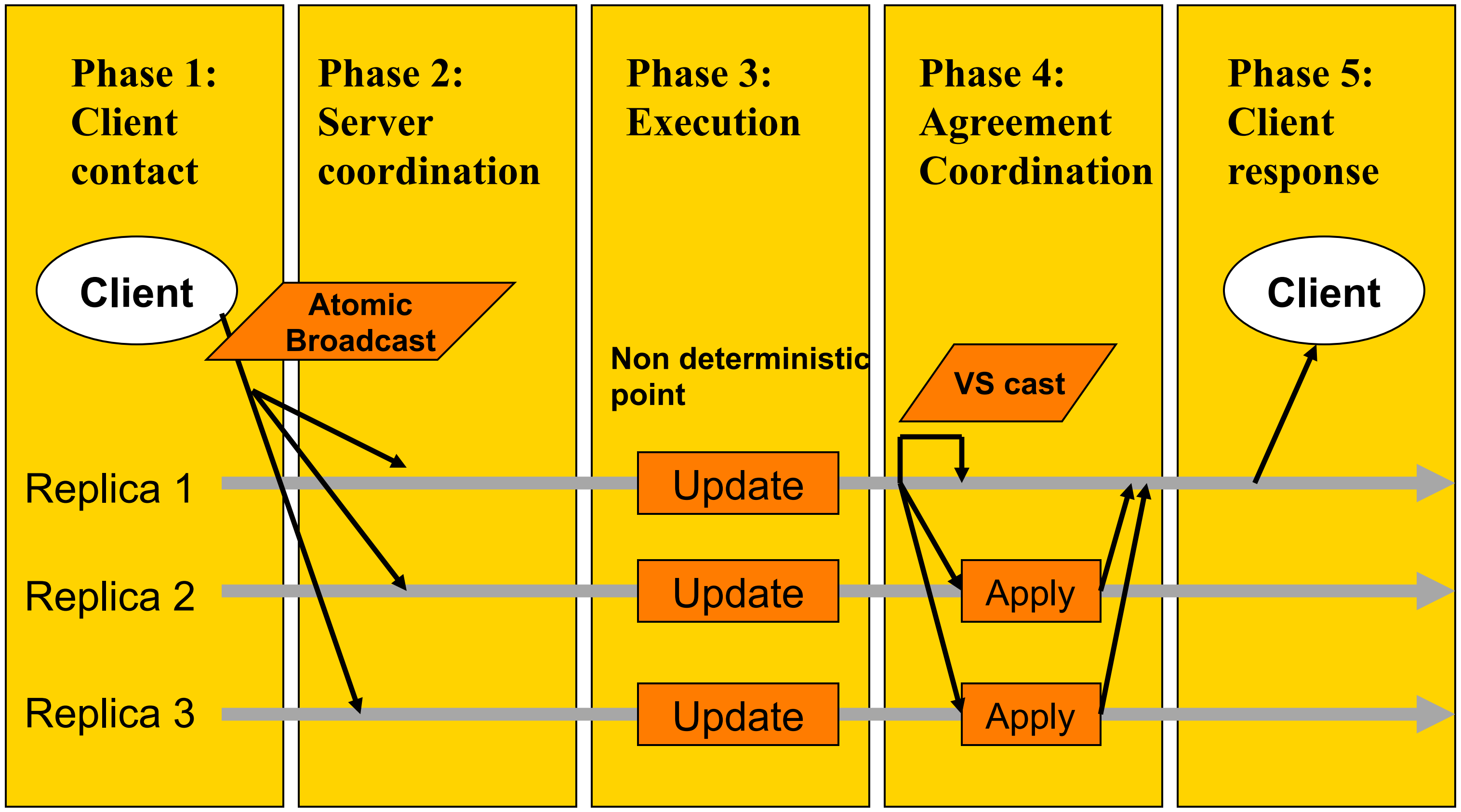


Replication in Distributed System

Semi-Active Replication (Weighted Voting)

1. By using an Atomic Broadcast the client sends the request to the servers.
2. The servers coordinate by using the order given by this Atomic Broadcast.
3. All replicas execute the request in the order they are delivered.
4. In case of a non deterministic choice, the leader informs the followers using the View Synchronous Broadcast.
5. The response is sent back to the client by the servers.

Semi-active Replication



Majority Consensus

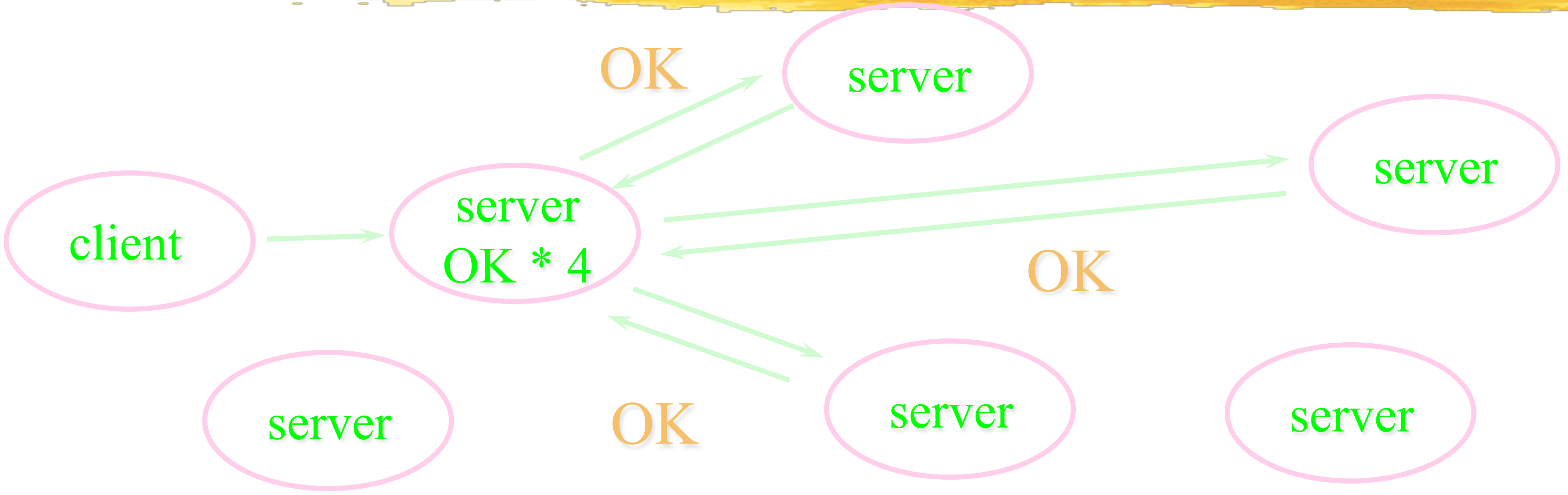
A request to be accepted and applied to all the representatives, only a majority of servers need approve it

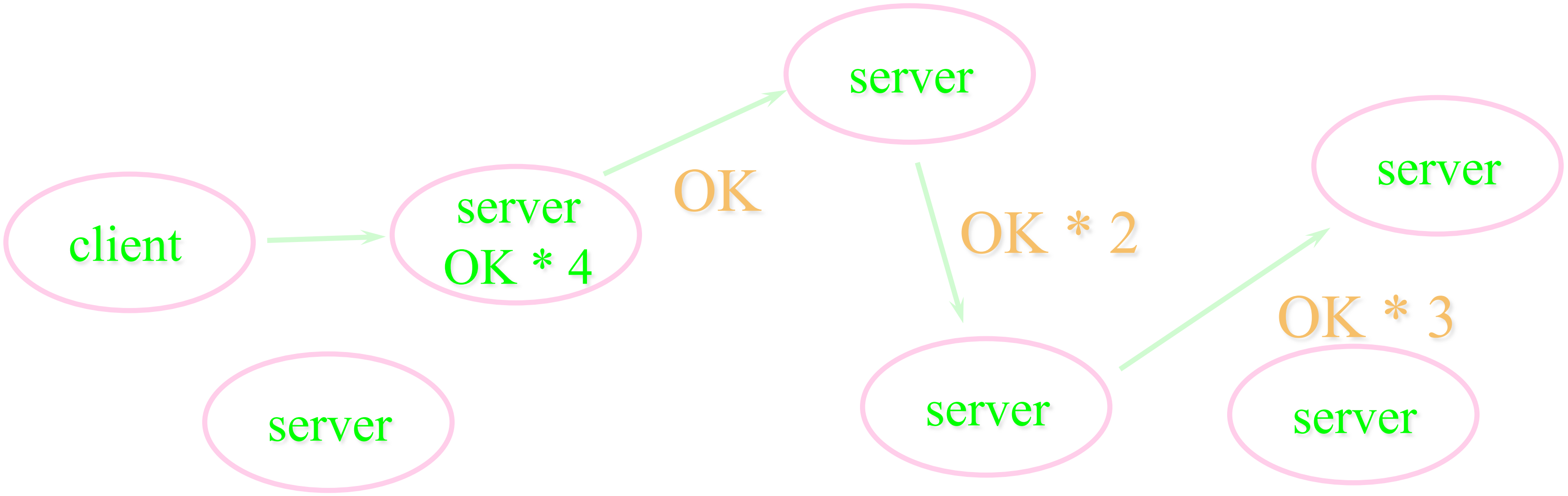
1. Clients request updates by submitting the old values, their timestamps, and the new values to one server

2. Each request is evaluated by servers using a voting rule

3. If request is accepted, each server subsequently does the update.

Majority Consensus (cont)





Majority Consensus (cont)

Replicated update request : R	
Operation	*Timestamp*
Read(a)	*tl*
Write(a-$2)	*tl*

Step	*Request*	*S1*	*S2*	*S3*
1	*Client -> S1:Read(a)*			
2	*Client -> S1:Write(a-$2)*	OK		
3	*S1 -> S2 : Request consensus*	OK		
4	*S2 votes OK*	OK	OK	
5	*S2 accepts update*	OK	DONE	
6	*S2 -> S1 do update*	DONE	DONE	
7	*S2 -> S3 do update*	DONE	DONE	DONE

Majority Consensus with Weighted Voting

Each representative has a number of votes stored in the file suite definition

Each read operation must first obtain a read quorum of r votes before it can proceed

Each write operation must obtain w votes before it can proceed (w > half)

if less than w can be found, non-current representatives are made current before continuing in the vote

Majority Consensus with Weighted Voting

File suite prefix P stored in each representative

UFIDs of representatives

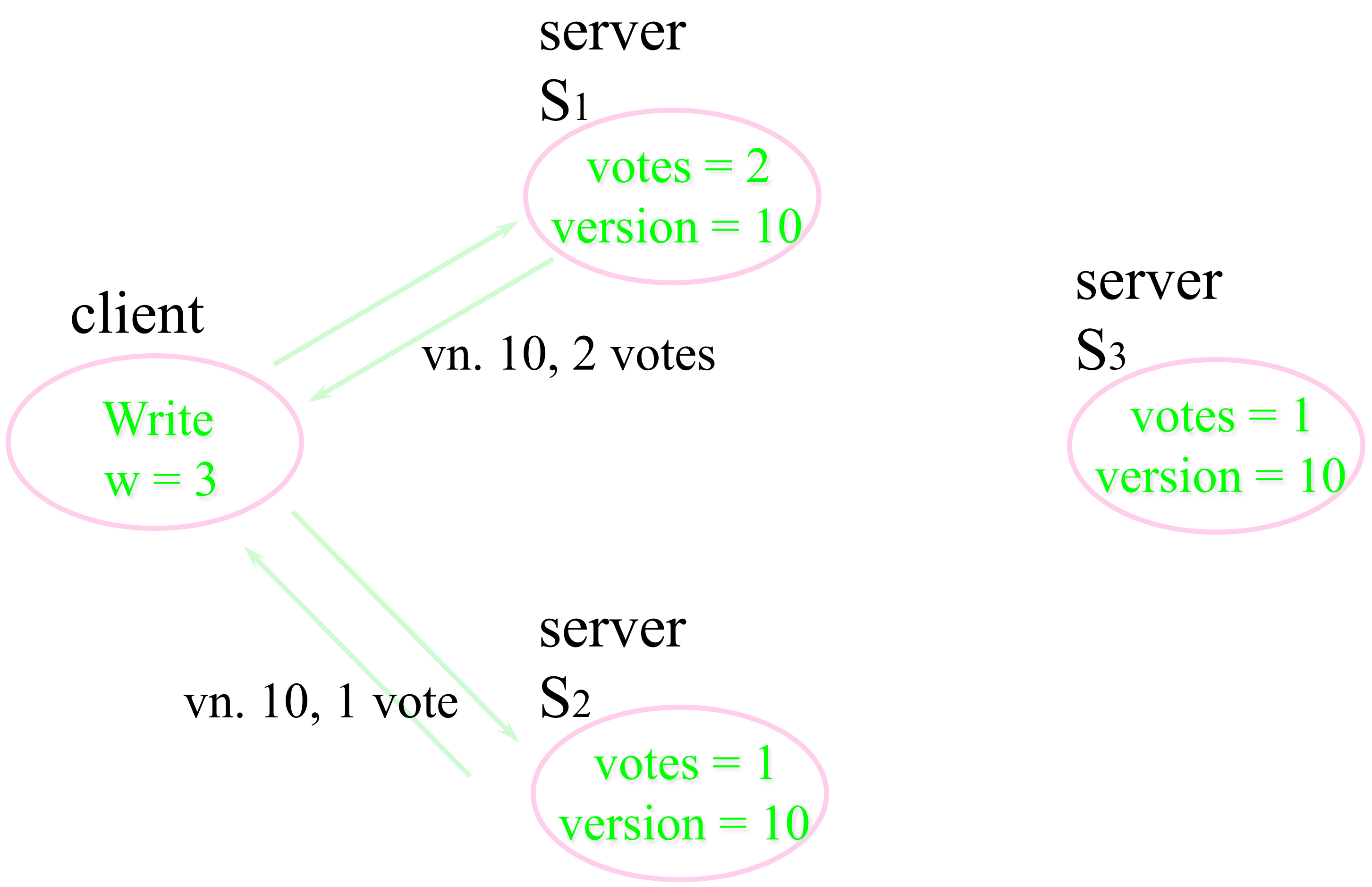
version number = 10		
r = 2		w = 3
UFID1	S1	2
UFID2	S2	1
UFID3	S3	1

votes of representatives

server ID's of servers with representatives

File suite prefix for Gifford's algorithm

Majority Consensus with Weighted Voting



Recovery

An important requirement for recovery is to have changes to data items be invisible to other transactions

Common Approaches:

1. Intentions list: list operations involved in a T, they will be executed at the end whenT is committed

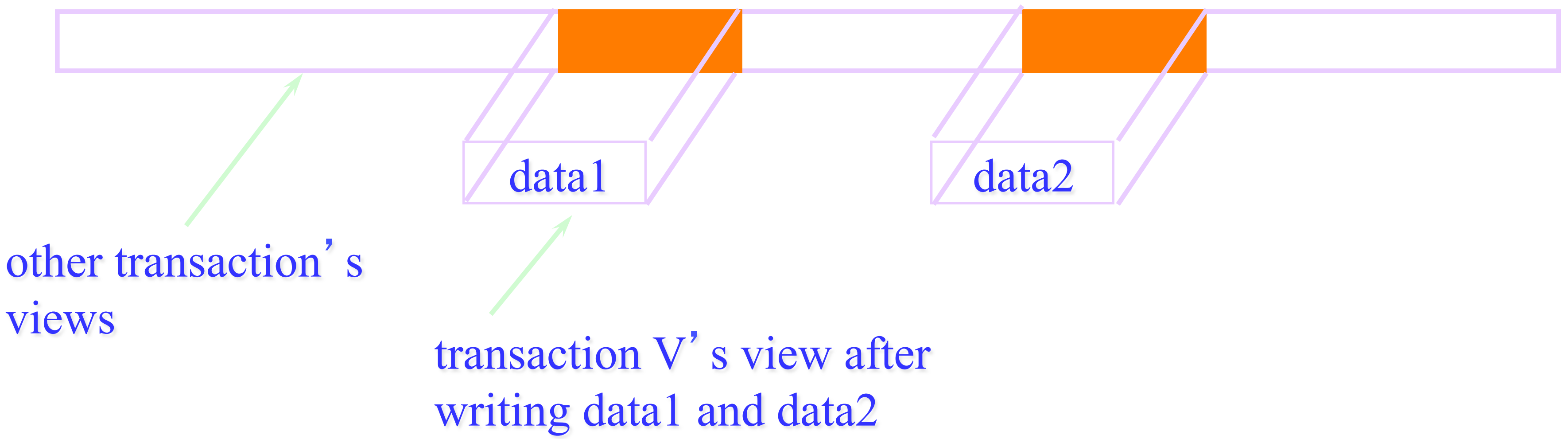
2. Version Approach: changes are stored in new versions which are discarded if T is aborted

Client operations	*Phase*	*Server actions*
OpenTransaction		
file access request	first phase	make tentative copies of changed items
CloseTransaction	*COMMIT*	
	second phase	incorporate tentative copies into files

Recovery (cont)

Representation of a file during phase 1 of a transaction

V = OpenTransaction;
Twrite(V, f, p1, data1);
TWrite(v, f, p2, data2);
CloseTransaction(V);



data1
data2
other transaction' s views
transaction V' s view after writing data1 and data2

The Intentions List Approach

Intention list maybe regarded as a log of operations

Commit flag indicates the state of a transaction

The intention list is kept in stable storage

Client operation	Phase	State	Intentions records
Open Transactiion			
Twrite(V, f, p1, data1);	first phase	tentative	{“write”, f, p1,data1}
TWrite(v, f, p2, data2);			{“write”, f, p2,data2}
Close TransactionCOMMIT			
	second phase	committed	intentions->files

The Intentions List Approach (cont)

- The commit flag is set to
  - tentative during the first phase
  - committed/aborted during the second phase
    - The changes made during the second phase must be performed atomically
- Transaction is completed when server executes all the operations in its intentions list successfully, regardless of crashes

  => this implies that these operations must be repeatable

The File Versions Approach

File is considered as a sequence of versions

When an operation modifies a file, the server creates a tentative version which becomes the current version when T commits

Conflicts during concurrent access of a file:

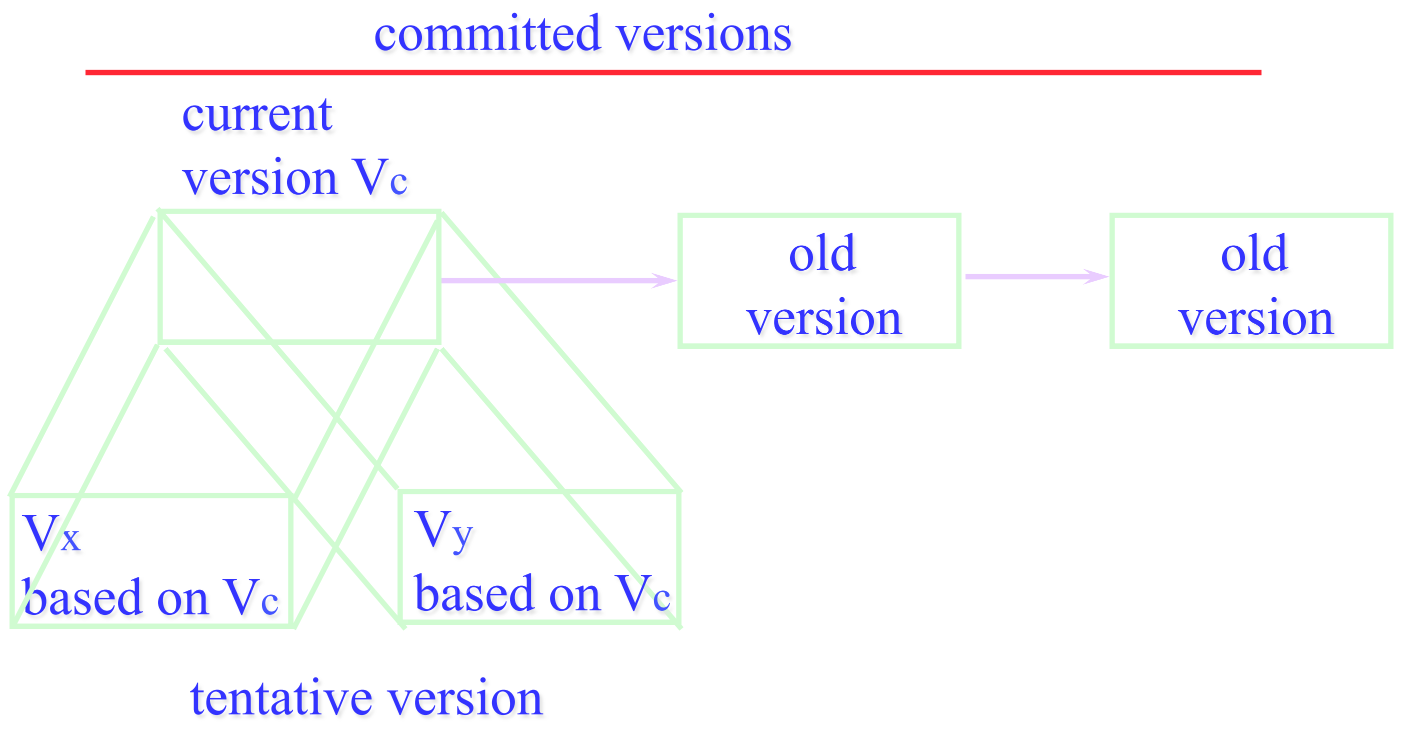
1. Version conflict
   - concurrent transactions modify distinct parts of the file (they do not access the same data)
   - merge action can be used to resolve this conflict
2. Serializability Conflicts
   - occurs when using optimistic concurrency control
   - concurrent transactions have accessed the same data item (some may modify it)
   - locking scheme can prevent these conflicts
   - timestamps can resolve them when they occur

Version Conflict Resolution

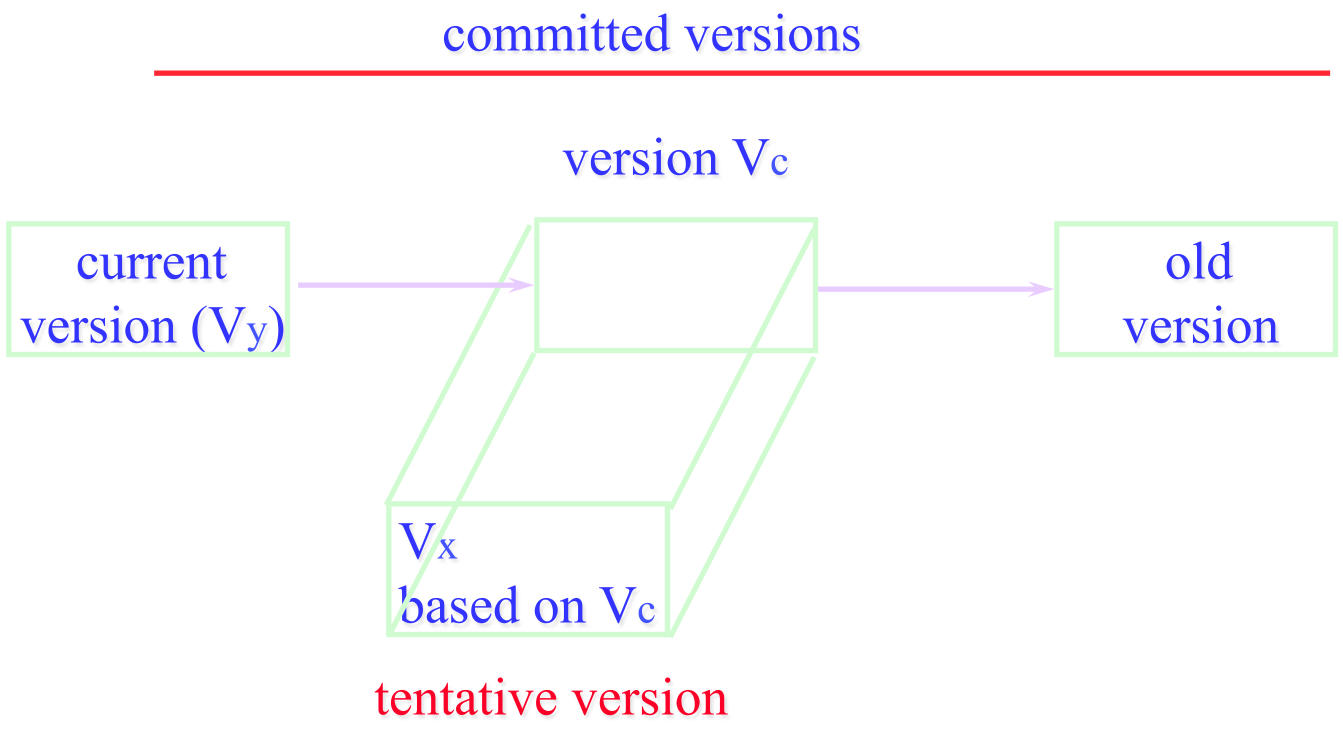
- In this scheme, server maintains a transaction record including a commit flag
- T-record contains the UFIDs of each tentative versions of modified files
- T-record is usually stored in stable storage in order to survive crashes
- When a transaction closes, there are several scenarios
- no serializability conflicts, tentative versions become current versions (the changes of more than one transactions can be merged when they close)
  - if conflict, all transactions involved in this conflict are aborted

Version Conflict Resolution (cont)

Representation of a file affected by two transaction X and Y



Version Conflict Resolution (cont)



Y has committed and X is about to commit. X will be aborted since it is based on an old version Vc

Distributed Transaction Service

Server need to coordinate their actions when the transaction commits in order to achieve recoverability and concurrency control

Coordinator / worker server :

- coordinator server is the one contacted first by the client. It is responsible for aborting or committing the distributed transaction

Each server has an intentions list to record all updates to local files from transaction

Distributed Transaction Service (cont)

Internal operations of distributed transaction service

NewServer(Trans, server-id, capability) -> ok

Call from a new server (in *AddServer*) to the coordinator. Caller supplies its *server-id* and a *capability*; the coordinator records *server-id* and *capability* in it worker list

CanCommit? -> yes / no

Call from coordinator to worker to check whether it can commit Worker replies with *yes / no*

DoCommit(Trans, capability)

Call from coordinator to worker to telll worker to commit its transaction

HaveComitted(Trans, server-id)

Call from worker to coordinator to confirm that it has committed transaction

Multi-server Commit

There is one coordinator that is responsible for commit/abort of the transaction when the client calls close transaction

Each server maintains its own transaction record, identifier of the coordinator server, and its intentions list

The *Close Transaction* is executed by the coordinator in two phases - preparing to commit and the commitment itself :

Phase 1 : Preparing to Commit

1. The coordinator has a list of the server-ids of the workers and its commit flag is *tentative;* it issues a *CanCommit?* call to each of the workers in the transaction with a timeout.

2. When the workers receive the *CanCommit?* call, each one does as follows

   If the value of its commit flag is *tentative* and the worker is able to commit (i.e. it has not previously aborted its part of the transaction), it returns *yes* to inform the coordinator it is ready.

   Otherwise the worker's commit flag is set to *abort* and it returns *no* as the result

Multi-server Commit

Phase 2 : Commit

3. At this point either the coordinator has received *ok* or *no* from each worker, or the *Can Commit?* call has timed-out.

- If the coordinator has had an *ok* return value from all of the workers in the transaction, it sets the value of its commit flag to *committed*, and makes the *DoCommit* call to all the workers. At this point, the transaction is effectively completed, since the coordinator and the workers are now committed to perform the updates in their intentions lists, so the coordinator can report success to the client.
- If any of the replies was *no* or any worker has been time-out, the coordinator sets its commit flag to *abort* and calls *AbortTransaction* in each of the workers, reporting failure to the client.

4. When a worker receives the *DoCommit* call it sets the value of its flag to *committed* and sends a *HaveCommited* call to the coordinator. It evetually carries out its intentions list and erases it.