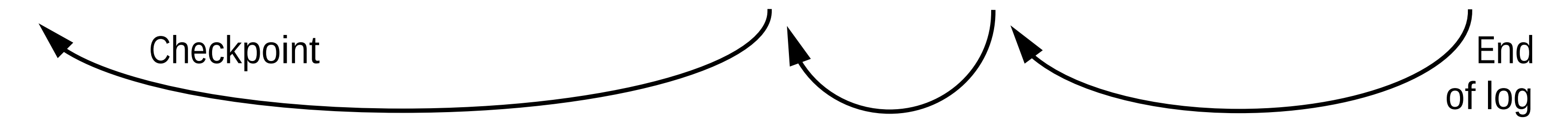
Entries in recovery file □ To deal with recovery of a server that can be involved in distributed transactions, further information in addition to the data items is stored in the recovery file. This information concerns the *status* of each transaction – whether it is *committed*, *aborted* or *prepared* to commit. In addition, each data item in the recovery file is associated with a particular transaction by saving the intentions list in the recovery file. To summarize, the recovery file includes the following types of entry:

Type of entry	*Description of contents of entry*
Data item	A value of a data item
Transaction status	Transaction identifier, transaction status (*prepared*, *committed*, *aborted*) – and other status values used for the two-phase commit protocol and for nested transactions (when in use)
Intentions list	Transaction identifier and a sequence of intentions, each of which consists of <identifier of data item>, <position in recovery file of value of data item>

Figure 15.1 Log for banking service.

P0			P1	P2	P3	P4	P5	P6	P7
Data:A 100	Data:B 200	Data:C 300	Data:A 96	Data:B 204	Trans:T prepared <A, P1> <B, P2> P0	Trans:T committed P3	Data:C 297	Data:B 207	Trans:U prepared <C, P5> <B, P6> P4



This technique is illustrated with the same example involving transactions T and U. The first column in the table shows the map before transactions T and U when the balances of the accounts A, B and C are \$100, \$200 and \$300. The second column shows the map after transaction T has committed:

Map at start	*Map when T commits*
A → P0	A → P3
B → P1	B → P4
C → P2	C → P2

	P0	P1	P2	P3	P4		
Version store	100	200	300	96	204	297	207

Checkpoint

Figure 15.2 Log with entries relating to two-phase commit protocol.

	Trans:T *prepared* Intentions list	Coord'r: T worker list: . . .	•	•	Trans:T *committed*	Trans:U *prepared* Intentions list	•	•	Worker:U Coord'r:...	Trans:U *uncertain*	Trans:U *committed*	

Figure 15.3 Recovery of the two-phase commit protocol.

Role	*Status*	*Action of recovery manager*
Coordinator	*prepared*	No decision had been reached before the server failed. It sends *AbortTransaction* to all the servers in the worker list and adds the transaction status *aborted* in its recovery file. Same action for state *aborted*. If there is no worker list the workers will eventually time-out and abort the transaction.
Coordinator	*committed*	A decision to commit had been reached before the server failed. In case it had not done so before, it sends a *DoCommit* to all of the workers in its worker list and resumes the two-phase protocol at Step 4 (see Figure 14.5).
Worker	*committed*	The worker sends a *HaveCommitted* message to the coordinator in case this was not done before the worker failed. This will allow the coordinator to discard information about this transaction at the next checkpoint.
Worker	*uncertain*	The worker failed before it knew the outcome of the transaction. It cannot determine the status of the transaction until the coordinator informs it of the decision. It will send a *GetDecision* to the coordinator to determine the status of the transaction. When it receives the reply it will commit or abort accordingly.
Worker	*prepared*	The worker has not yet voted and can abort the transaction.
Coordinator	*done*	No action is required.

Cristian [1991] provides a useful classification of failures. A request to a server can change the state of its resources and may produce a result for the client. Cristian's classification assumes that for a service to perform correctly, both the effect on a server's resources and the response to the client must be correct. Part of the classification is given in the following table:

Class of failure	*Subclass*	*Description*
Omission failure		A server omits to respond to a request
Response failure		Server responds incorrectly to a request
	Value failure	Returns wrong value
	State transition failure	Has wrong effect on resources (for example, sets wrong values in data items)

Examples of Faults

- Omission Failure
  - UDP
- Response Failure
  - At once RPC semantics masks omission failures but may convert faults into response failures if service is not idempotent

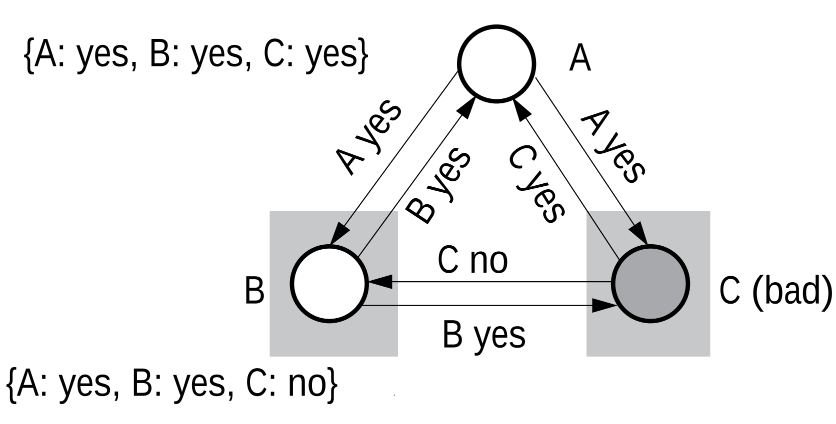
An important aspect of a server failure is its state after it has been restarted. For example, a transactional service restarts with the effects of all committed transactions reflected in its data items. Cristian gives the following classification of server failures:

Class of failure	Subclass	description
Crash failure		Repeated omission failure: a server repeatedly fails to respond to requests until it is restarted
	Amnesia-crash	A server starts in its initial state, having forgotten its state at the time of the crash
	Pause-crash	A server restarts in the state before the crash
	Halting-crash	Server never restarts

Failure Semantics

- Fail-stop services
- Byzantine failure
  - Byzantine general's problem
    - If message originators can be authenticated, 2N+1 servers can tolerate N faulty servers
    - If no sender authentication, need at least 1/3 of the participants to be non-faulty

Figure 15.5 Byzantine Generals.



(a) Message originators can be authenticated by receivers

(b) Message originators cannot be authenticated by receivers

Masking of Faults

- Hierarchical
  - Server at higher level masks faults at lower level
- Group failure masking
  - Closely synchronized group of servers
    - Each replica executes on a different computer and executes same requests
  - Loosely synchronized group of servers
    - Primary server + backup servers

Figure 15.6 Three-way message from A to B.