Distributed Systems

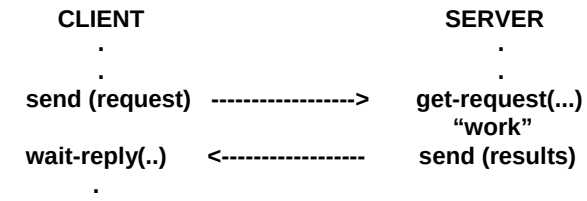
Paul D Ezhilchelvan

1. Client-Server Interactions

rpc1

(see chapts. 10.2.6+, Modern Op Sys, or chapts. 2.3.7+ Dist Op Sys, Tanenbaum)

- Typical Interaction: Client (user process) sends a request message to a server process and blocks, awaiting a reply; the server performs the requested task and sends the reply to the client who is then unblocked:



rpc2

- Issues to discuss:

1. Protocols
   - Message fragmentation and reassembly: application level messages may have to be fragmented into a number of smaller messages (at sender) and reassembled (at receiver), because underlying networks impose limitations on the max. message (packet) size (typically a few thousand bytes).
     - we will not discuss fragmentation and reassembly protocols and assume such a protocol has been implemented, allowing sending and receiving of arbitrary length messages .
   - Types of messages required
2. Binding and integration with languages
   - locating servers
     - static binding, dynamic binding
   - Remote procedure Call (RPC) way of integration with a language
     - client and server stubs
     - parameter passing
     - parameter marshalling and unmarshalling

rpc3

3. Failures and their impact on RPC semantics

- A quick look at the problems:
  - client crashes after making a request
    - the server becomes an 'orphan'
  - server crashes: client could wait for ever
  - request (reply) message gets lost: client could wait for ever
  - client suspecting lost message, resends the request: server could end up executing the request more than once.

rpc4

- **Protocol Issues: message types** (in the table below, C: client, S: server)

Code	Type	From	To	Remarks
1. REQ	request	C	S	Client wants service
2. REP	reply	S	C	Reply from server to client
3. ACK	ack	S/C	C/S	Previous msg arrived
4. AYA	are you alive?	C	S	Enq. msg to check if server is functioning
5. IAA	I am alive	S	C	Server responding that it is functioning
6. TA	try again	S	C	Server is busy, has no room (eg, no buffer space)
7. AU	address unknown	S	C	No process is using this address

rpc5

1-2: message types 1 and 2 are essential.

3: added to increase reliability (cope with message loss).

4-7: not necessary for basic service, but useful for additional functionality

Need for 4-5: Consider that the client has sent a request. What if no reply coming after a reasonable time? Is the server still performing the work or is there a breakdown (eg server has crashed)?

Client uses AYA message for 'probing' the server; if the reply is an IAA msg (or REP), then all is well; else if even after a few AYA probe messages, no IAA/REP message is forthcoming, client can be reasonably sure that the server is unavailable.

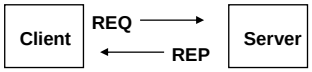
6: There are cases when the server is unable to accept a REQ msg.

- Server machine has no more buffer space for storing a new message (server has received many requests, which have consumed the available buffers).
- Server is overloaded, so is refusing further requests for a while.

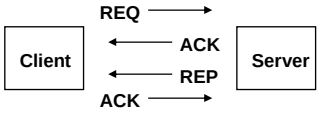
7: Wrong address (there is no server process with this address at that node), so retrying by client will not help.

rpc6

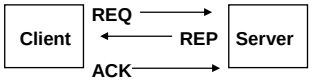
- Examples



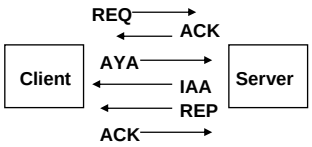
SIMPLE PROTOCOL WITH NO ACK



PROTOCOL WITH ACK FOR EACH MSG



PROTOCOL WHERE REP ALSO ACTS AS ACK



PROTOCOL WITH ACK AND PROBES

rpc7

- **Remote procedure Call (RPC):**

  A way of encapsulating client-server interactions into a simple (to the user) operation such that:

  (i) client: invokes a service call and gets the results;

  (ii) server: accepts the request and executes the called procedure.

- **Language specific tool (a stub compiler) can be provided to hide details of message passing.**
- **First consider how a local call is made to a precompiled routine (eg, a library routine):**

  Let S be a routine (a procedure) and U be a user program that contains a call to S:

  S has been compiled: S ---> compiler --> S_{obj} (object code)

  U is compiled: U --> compiler --> U_{obj} (object code)

  U_{obj} is then linked to S_{obj} using a linker/loader and then executed.

rpc8

- **To invoke S remotely, we proceed as follows.**
  - **U_{obj} is linked to a client stub for S that is responsible for using the underlying protocol for sending the relevant request message and then collecting the result message;**
  - **S_{obj} is linked to a server stub for S that is responsible for receiving a request, calling S and then sending the results back to the caller as a message.**
  - **The stubs can be produced using the interface definition of S (eg, the class definition):**

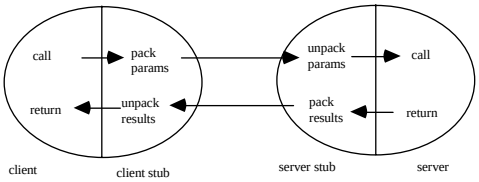
Let S_{if} be the interface definition of S

S_{if} -- Stub Compiler - → **C_{stub} (client stub)**

S_{if} -- Stub Compiler - → **S_{stub} (server stub)**

U_{obj} is linked to C_{stub} and S_{obj} is linked to S_{stub}

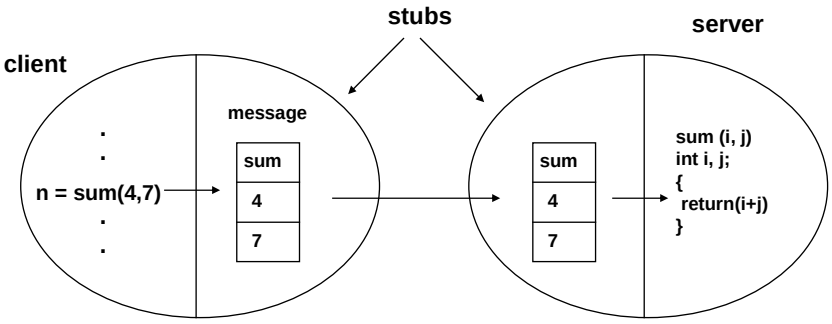
rpc9



1. the client calls the client stub in the normal way;
2. The client stub builds the call message, marshalling (packing) the parameters, and calls the operating system routine for message sending;
3. The operating system sends the message to the server node;
4. The server node receives the message, hands it to the server stub;
5. The server stub unmarshalls (unpacks) the parameters, and calls the actual routine;
6. The call is executed and the results returned to the server stub;
7. The server marshalls the parameters, builds the result message and hands it to the operating system for transmission;
8. The message is sent to the client node;
9. the client stub receives the message;
10. The message is unmarshalled, and the stub returns the call to the client.

rpc10

- **example:**



rpc11

- **RPC Design Issues:**
  - **Parameter Passing**
  - **Binding (locating servers)**
  - **Treatment of failures**
- **Parameter Passing**
  - **languages employ various techniques. For example, C uses call by value and call by reference.**

Explanation of parameter passing techniques:

(a) Call by value: The parameter value is copied on the stack; in the called procedure, a local variable gets initialised to this value. The called procedure may modify it, but such changes do not affect the value of the original variable.

(b) Call by reference: The reference to the variable (pointer, address of the variable) is copied on the stack; the called procedure can therefore access the variable and changes are directly reflected in the calling program.

(c) Call by value-result (call by copy-restore): The variable's value is copied on the stack as in (a), then copied back on the stack after the call, overwriting the original value. (used in Ada).

rpc12

- **Under most conditions, call by value-result has the same effect as call by reference.**
- **In RPCs, how should call by reference parameters be treated?**
  - **Address in one machine is not meaningful in another. So, the actual object, rather than the reference to it should be passed.**
  - **Use call by value-result in place of call by reference.**
- **Parameter packing and unpacking:**

Packing parameters into a message is called parameter marshalling and unpacking the message into the parameters is called unmarshalling.

Machines can have different ways of representing numbers, strings etc.

For example, in Intel chips, bytes of a word are numbered from right to left, whereas on Sun Sparc chips, it is left to right:

Integer value 5 stored in a 4 byte word at Intel chip will be received at Sparc chip as 5000 (= $5 \times 2^{24} = 83886080$)

- It is therefore necessary to have a standard way of representing basic data types:if parameters are marshalled as per the rules of this standard, then they can be unmarshalled at the server independent of the byte order used at the client machine.
  - This does mean that marshalling and unmarshalling can be a bit inefficient (because parameters need to be converted into standard representation for packing and then reconverted into machine specific representation at the server)
  - read Chapter 2.4.2 (or Chapter 10.3.2) of Tanenbaum book for more details

- **Static and Dynamic Binding**
  - So far we have not discussed how a client locates the server.
  - One method is to write the client program (generate the client stub) for a specific server at a specific machine.
    - This is static binding (simple, but not flexible).
- **Dynamic binding: locating servers at run time.**
  - The following steps are undertaken:

  (i) The specification of the service is stored (registered) with a binder (a name server)
    - binder is a server process running at a well-known address
    - the service specification includes:
      - name of the service
      - service routine specification (interface definition or class definition)
      - address (host name, port name) of the server running that service

    The act of registration is also called exporting

rpc15

(ii) The server (service provider) uses the service specification for generating the server stub and linking it to the service routine (the server is ready to receive calls)

(iii) Application builder can examine the service specifications registered in the binder (using some browser) for the desired service, and select a service specification, pass it through stub generator for creating a client stub;

- the client stub has code for contacting the binder at run time for obtaining the address of the server by presenting the service name.
- Thus the client program does not need recompiling even if the address of the server changes.

- **A specification language is necessary for specifying service routines**
  - For C++ programs, one can use C++ class definitions , but this becomes rather language specific. Special Interface Definition Languages (IDLs) have been suggested to allow mixed-language programming
    - you need IDL to your language stub compiler
    - so, client and servers could be written in different languages.

rpc16

Example of an IDL specification for a file read routine:

read (in char name[max-path], out char buf[buf-size], in int no-of-bytes,
in int position)

in/out specify the direction of information flow (relative to the server).

Here, the server is expecting a path name (file name), no. of bytes to be read from position and results to be put into buf.

Client sends in parameters to the server in the call request, the server's message will contain out parameter.

rpc17

- **Treatment of failures and RPC semantics**

- **Failure Model:**
  - **(i) Communication failures: functioning processes temporarily unable to communicate**
  - **(ii) Node crashes: a node (computer) either works or crashes (stops working); a crashed node eventually recovers.**

- **RPC Semantics:**
  - **We first need to define normal and abnormal termination of a call.**
  - **Normal Termination: client receives the reply message.**
  - **Abnormal (exceptional) termination: client does not receive the reply (ie receives no message at all or an address unknown (AU) message).**

- **We now define three RPC semantics by considering the meaning of a call under normal and abnormal terminations.**

rpc18

(i) Exactly Once
- **normal termination: the call was executed once.**
- **abnormal termination: the call was not executed.**

(ii) At most once
- **normal termination: the call was executed once.**
- **abnormal termination: the call was not executed or executed partially or executed once.**

(iii) At least once
- **normal termination: the call was executed one or more times.**
- **abnormal termination: the call was not executed or executed partially or executed one or more times.**

- **Remarks:**
  - **Exactly once: most desirable, but difficult to implement.**
  - **At least once: easy to implement, but not desirable (however, it is perfectly acceptable in situations where multiple executions are same as a single execution, eg., read operation)**
  - **At most once: a compromise solution that is relatively easy to implement**

rpc19

- **Understanding how failures affect RPC semantics**
- **First, we begin by defining the possible failure events during a call:**
  1. **Client is unable to locate server process**
  2. **Client's request message lost**
  3. **Server's reply message lost**
  4. **Server crashes during the call**
  5. **Client crashes during the call**
- **Second, we assume the following actions at the RPC protocol level:**
  - **Event 1: client gets 'address unknown' message from the called node. The call is terminated abnormally.**
  - **Events 2, 3, 4: client sets a timer after sending the request message, so the client never blocks.**
    - **Client's response to the timeout:**
      - **terminate the call abnormally, or**
      - **client retries by re-sending the request message**
        - **retries are performed a finite number of times, after which the call is terminated abnormally**

rpc20

- **RPC semantics in the presence of communication failures**

- Client sends a request, and then timesout, waiting for the reply

- **Client does not retry: RPC semantics is *???***
- **Client retries till a response is obtained.**
  - **Response is AU, call terminated abnormally.**
    - **RPC semantics is *????***
  - **Response is the reply.**

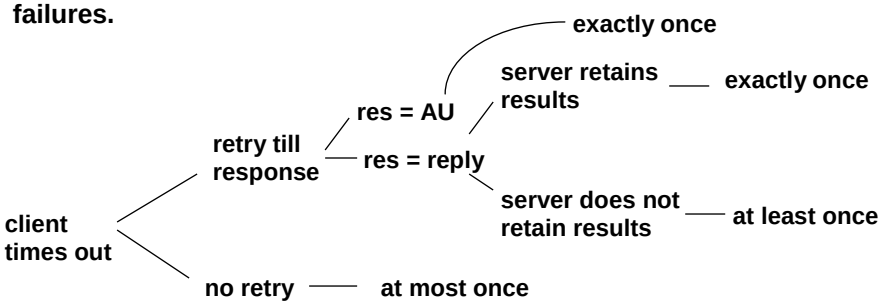
    There are two cases to consider:
  - **Case I: Server does not maintain any state information about previous executions; so the server can end up executing the call more than once (as a retry message is treated as a fresh call request).**
    - **RPC semantics is *?????***
  - **Case II: Server maintains the results of the last execution in a buffer; if a request message is received, and is recognised as a duplicate (retry), it re-sends the the results.**
    - **RPC semantics is *?????***

- Client sends a request and receives AU

- **RPC semantics is *?????***

rpc21

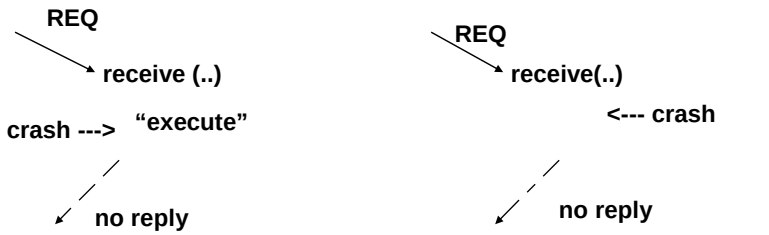
- **Summary: RPC semantics in the presence of communication failures.**



Remark: If the only failures are communication failures, then it is not difficult to implement exactly once semantics.

rpc22

- **RPC semantics in the presence of server crashes**
- **Two cases to consider:**



crash during/after execution **crash before execution**

Client has no way of finding out what has happened.
Client will timeout, waiting for the reply.

- **Client's response to timeout:**

(i) Keep on trying (resending the request): eventually a response will be received (as the crashed machine eventually recovers)

- **response will be either the reply or address unknown: RPC semantics is *at least once*.**

(ii) Terminate the call: RPC semantics is *at most once*.

- **RPC semantics in the presence of lost messages and server crashes:**
  - **We now combine the results gained from the previous discussions (RPC in the presence of comms failures and RPC in the presence of server crashes) to develop RPC protocols that provide specific semantics in the presence of lost messages and server crashes.**
- **<u>At Least Once RPC</u>**
  - **<u>Client</u>: Client makes a call with a timeout. If the timer expires (because no response from the called server has been received), then the client retries. Retries are made a finite number of times.**
  - **<u>Server</u>: Server executes the received call request and sends the reply; server does not maintain any state information on previous calls (such a server is called a <u>stateless server</u>)**

rpc25

- **At Most Once RPC**
  - **Client: Client retries as before. Call request messages contain call sequence numbers, such that a retry request message has the same sequence number as the main call request message, and a new call has a sequence number that is greater than the previous one.**
  - **Server: For each of its clients, the server maintains the results of the last executed call and the call message sequence number (so the server is no longer stateless).**
    - **Each call request is checked for 'freshness': if it is a retry, then the server resends the stored results, else the call is executed.**
    - **We assume that a crash of the server destroys the above state information. So how do we ensure that after recovering from a crash, the server can detect (pre-crash) retry messages?**
    - **Answer:**

rpc26

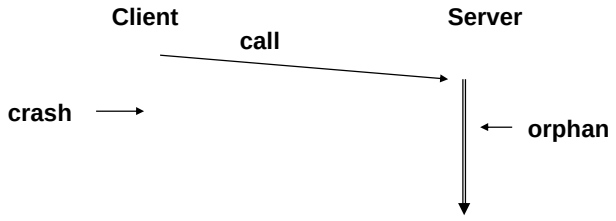
- **Exactly Once Semantics**
  - **What is required is 'all or nothing' behaviour:**
    - **normal termination: one execution,**
    - **abnormal termination: no execution.**
  - **Difficult (if not impossible) to achieve in the presence of server crashes. Server could be in the middle of performing an 'unrecoverable action' , such as printing, drilling a hole....**
  - **If server actions are recoverable (changes to data), then crash recovery actions can be undertaken to 'undo' effects of any partial executions, and create the illusion that no execution took place.**
    - **requires client and servers to take coordinated recovery actions otherwise strange behaviour is possible:**
      - **suppose server sends results to client and then crashes; crash recovery of server restores pre-crash state, but the client has received the results. No execution as far as the server is concerned, but execution as far as the client is concerned.**

rpc27

- The general approach is to use Atomic Transactions (also called atomic actions). Transactions make use of special coordination protocols between client and servers to ensure 'all or nothing' property.
  - Subject of next few lectures.....
- So, the practical approach is to implement at most once RPCs, and make use of atomic transactions at the application level to ensure consistency of applications despite lost messages and crashes.

rpc28

- Client Crashes
  - If a client crashes after making a call request which is now being executed by the server, then that execution is called an orphan execution (orphan for short):

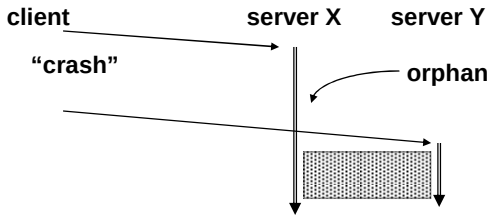


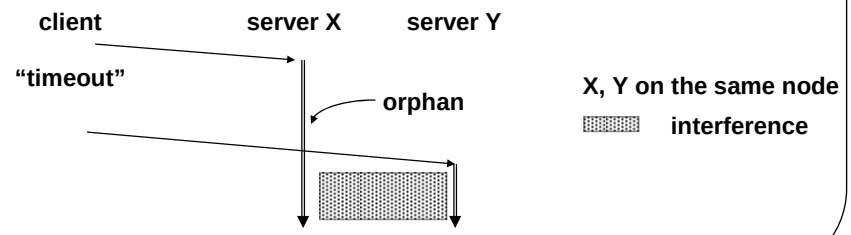
Orphans consume resources (processing power, memory buffers,..). Suppose, in the above example, client first locks a file at the server, but does not unlock it (as the client crashes); then, the file will remain locked.

Orphans can create consistency problems by interfering with normal executions.

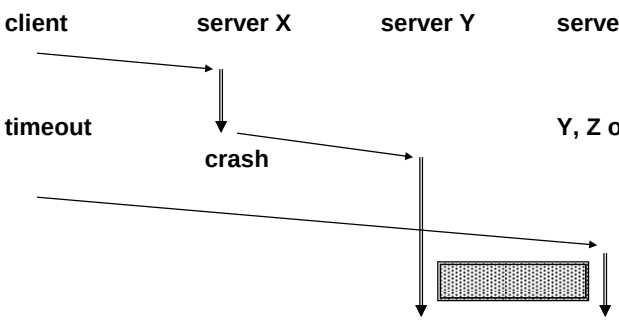
rpc29

- **Interference caused by orphans: consider the execution of a single sequential program.**





rpc30



In the above example, the client timeout waiting for the reply from X; its subsequent action consists of making a new call to server Z.

- **Orphan treatment**

  (i) Server, upon receiving a new call request, should check that no previous call request from the same client is in execution on that node; if so, it should abort/terminate the previous call request, before executing the current request.

  (ii) A server node should periodically check that its clients are 'alive'. If a failure of a client is suspected, then it should abort/terminate any orphans.

Time, Clocks and Ordering of Events

T1

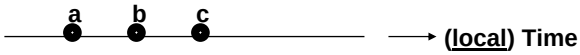
- **In a distributed system, it is often necessary to be able to establish relationships between events occurring at different processes:**
  - **was event 'a' at P_1 responsible for causing 'b' at P_2?**
  - **is event 'a' at P_1 unrelated to 'b' at P_2?**
- **We will discuss the partial ordering relation "happened before" defined over the set of events.**

T2

- **Assumptions:**

(i) Processes communicate only via messages

(ii) Events of a process form a sequence, where event 'a' occurs before 'b' in this sequence if 'a' happens before 'b' (Note: events of a process are totally ordered):



(iii) Sending or receiving a message is an event

- **We define the 'happened before' relation, denoted by '$\rightarrow$' as follows:**

T3

- **Definition : The relation '$\rightarrow$' on the set of events of a system satisfies the following three conditions:**

  (i) if 'a' and 'b' are events in the same process, and 'a' comes before 'b' then a $\rightarrow$ b

  (ii) if 'a' is sending of a message by one process and 'b' is the receipt of the same message by another process, then a $\rightarrow$ b

  (iii) if a $\rightarrow$ b and b $\rightarrow$ c then a $\rightarrow$ c

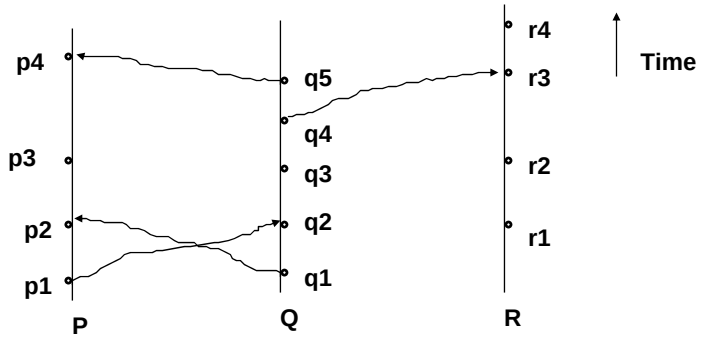
  Being meaningful, we assume a $\not\rightarrow$ a// irreflexive

- **Definition: Two distinct events 'a' and 'b' are said to be concurrent (a || b) if a $\not\rightarrow$ b and b $\not\rightarrow$ a.**
- **So, $\rightarrow$ defines a partial order over the set of events**

  Since there could be concurrent events in the set, that by definition are not related by $\rightarrow$.
- **Another way of regarding 'a $\rightarrow$ b' is to say that 'a' causally affects 'b'**

T4

- **Space time diagram**



a $\rightarrow$ b : one can go from a to b in the diagram moving forward in time along the process and message lines.

p1 $\rightarrow$ r4, q4 $\rightarrow$ r3, p2 $\rightarrow$ p4, q3 || p3, q3 || r2

T5

- **Problem: How to implement an event numbering scheme which respects the ' $\rightarrow$' relation. We do this by using logical clocks.**

- **Logical Clocks:**
  - **We introduce clocks into the system . We begin with an abstract point of view in which a clock is just a way of assigning a number to an event.**
  - **We define a clock C_i for each process P_i to be a function which assigns a number $C_i(a)$ to 'a' in P_i.**
  - **The entire system of clocks is represented by the function C which assigns to any event 'b' the number C(b), where $C(b) = C_j(b)$ if 'b' is an event in P_j.**
- **How to specify the correctness condition for these clocks?**
  - **Since our clocks are not measuring the passage of time, we cannot base our condition on physical time. We use the relation $\rightarrow$ for stating the correctness condition:**
- **<u>Clock Condition</u>:**
  - **for any events a, b: if $a \rightarrow b$ then $C(a) < C(b)$**

T6

- **Note that we cannot expect the converse of the clock condition to hold as well**
  - **we cannot say that if $C(a) < C(b)$ then $a \rightarrow b$, since this would require two concurrent events to occur at the same logical time (have the same clock value).**

- **The clock condition can be satisfied if the following two conditions hold:**

  CL1: if a and b are events in P_i and $a \rightarrow b$, then $C_i(a) < C_i(b)$.

  CL2: if a is the sending of a message by P_i and b is the receipt of that message by P_j, then $C_i(a) < C_j(b)$.

T7

- **Implementing logical Clocks:**

  Each process (P_i) has a register (C_i) and $C_i(a)$ is the value contained in C_i during event a.

- **<u>Implementation Rules</u>:**

  <u>IR1</u>: each process P_i increments C_i immediately after the occurrence of a local event

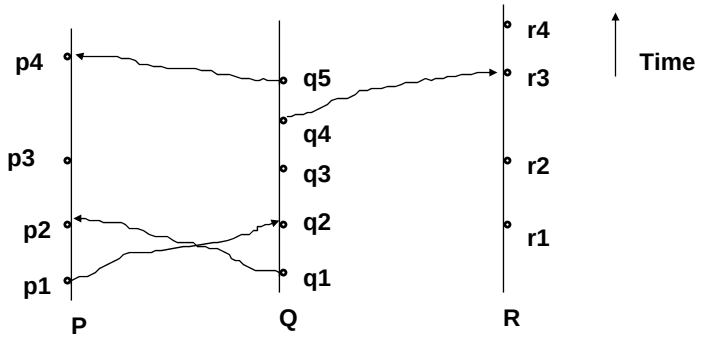
  - **IR1 meets the condition CL1. To meet condition CL2, we require that each message contain a clock value, Tm (called the timestamp)**

  <u>IR2</u>: (i) if 'a' is an event representing the sending of a message 'm' by P_i to P_j, then m contains the the timestamp $Tm = C_i(a)$

  (ii) receiving m (event b) by process P_j : peek at m; increments C_j, if necessary, to ensure that $C_j(b) > Tm$; execute receive(m).

T8

- **Space time diagram**

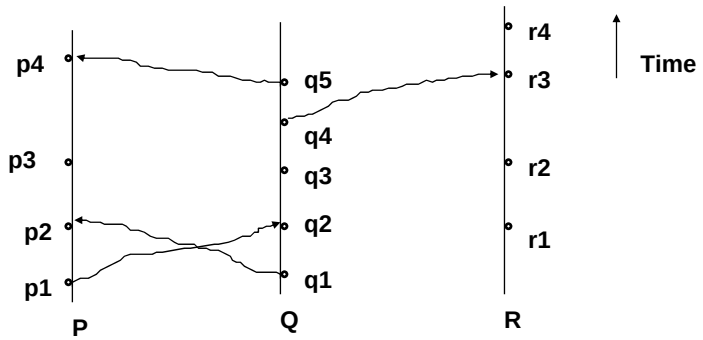


a → b : one can go from a to b in the diagram moving forward in time along the process and message lines.

p1 → r4, q4 → r3, p2 → p4, q3 || p3, q3 || r2

T9

- **Space time diagram**



$a \rightarrow b$: one can go from a to b in the diagram moving forward in time along the process and message lines.

$p1 \rightarrow r4$, $q4 \rightarrow r3$, $p2 \rightarrow p4$, $q3 \parallel p3$, $q3 \parallel r2$

T10

- **<u>Ordering Events Totally</u>**

  It is often useful to be able to put a total order on the set of all events. We simply order events by the logical time assigned to them. To break ties ($C_i(a) = C_j(b)$), we impose some fixed rule. One such rule could be based on process numbers (assume each process has a unique number).

- **We define a relation '$\Rightarrow$' (total order) as follows:**

  $a \Rightarrow b$ (read a ordered before b) if and only if $C_i(a) < C_j(b)$,

  or,

  $C_i(a) = C_j(b)$ and $P_i < P_j$.

  Notes: (i) if $a \rightarrow b$ then $a \Rightarrow b$;

  (ii) while $\rightarrow$ is unique for a given set of events, there could be several $\Rightarrow$ orderings, depending upon the rule used to break the ties, and upon the way individual process implements IR1.

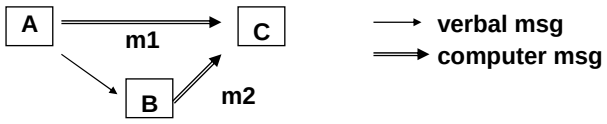
  (iii) total ordering based on logical clocks cannot be guaranteed to respect the temporal ordering of events that are not related by $\rightarrow$; this can give rise to anomalous behaviour.

T11

- **Anomalous behaviour:**

  A system S using logical clocks for total ordering can permit anomalous behaviour, whereby events are observed to happen in an order that is not consistent with the ordering indicated by the clock numbers.

- **Example:**
  - **Suppose you send a message m1 (event a) from computer A to computer C and then verbally instruct your colleague on computer B who then sends a message m2 (event b) to computer C. Clearly, b has happened after a, but the timestamp of m2 cannot be guaranteed to be greater than that of m1.**



The reason is that in S, a and b are not related by $\rightarrow$, as verbal messages are not part of S.

How can anomalous behaviour be prevented?

T12

- **Let ➔ be the happened before relationship in system S which includes all events of interests (including those of S and sending/receiving of verbal messages). We require the following strong clock condition to prevent anomalous behaviour in system S:**

  For any events a, b in S: if a ➔ b then C(a) < C(b)

- **The strong clock condition cannot be met by logical clocks (as they 'tick' only for events in S). However, if clocks can be made to represent physical time, then the strong clock condition can be met. So we introduce physical clocks in our system.**

- **Physical Clocks:**

  Let $C_i(t)$ denote the reading of the clock C_i at physical time t. For mathamatical convenience, we assume that clocks run continuously, rather than in discrete 'ticks'.

T13

$dC_i(t) / dt$ represents the rate at which the clock is running at time t.

A non-faulty physical clock must run at approximately correct rate (unfortunately , a clock cannot be absolutely correct).

so, $dC_i(t) / dt - 1$ for all t. We state this as a correctness condition PC1:

PC1: There exists a constant k ($k << 1$), such that for all t:

$$| dC_i(t) / dt - 1 | < k$$

(for a crystal clock, $k \approx 10^{-6}$)

That is a clock C_i will measure a time interval d as:

$$d - k\,d < C_i(t+d) - C_i(t) < d + k\,d$$

↑ slow clock ↑ fast clock

It is not enough for the clocks to individually run at approximately correct rate, they must be synchronised such that $C_i(t) - C_j(t)$ for all t. We state this as correctness condition PC2. Let ε, denoting synchronisation error, be a small constant such that:

PC2: for all i, j: $| C_i(t) - C_j(t) | < \varepsilon$

- **How small k and ε should be to prevent anamolus behaviour?**

T14

Let μ be the minimum message transmission time in S. That is, if a → b, and a is a send event at time t, then the receive event b will happen after $t + \mu$.

Note: μ could be the physical distance divided by the speed of light; in practice, it could be significantly larger.

- **To avoid anomalous behaviour, we want that:**

for any i, j: $C_i(t+\mu) - C_j(t) > 0$ (i)

From PC1, $C_i(t+\mu) - C_i(t) > (1 - k)\mu$

From PC2, $C_j(t) < C_i(t) + \varepsilon$

Restating (i): we want $C_i(t+\mu) > C_j(t)$.

$C_i(t+\mu) \geq C_i(t) + \varepsilon$ (using PC2)

Therefore, $C_i(t+\mu) - C_i(t) \geq \varepsilon$

so, $(1 - k)\mu \geq \varepsilon$ (by PC1)

That is, the minimum transmission time μ, as measured by a slow clock should be greater than or equal to the clock synchronisation error ε.

Synchronising Physical Clocks

T15

- We have seen the need to have small ε - synchronisation error
- Physical clocks do drift apart, due to non-identical *k*
- So, Periodically their readings must be adjusted to keep ε small

Two Problems to be solved:

(i) how to adjust the clock reading

(ii) by how much

How To:

Hardware Approach:

- adjust the running rate directly while crystal oscillations are being turned into clock 'tick's

Software Approach:

- physical clock is a read-only register
- hence construct an abstraction: synch_clock = physical_clock + Adj, where Adj is a register containing the necessary adjustment at any given time.
- updating Adj should not be discreet, but be gradual over a period of time.
  - jumps or rollbacks in clock readings should be avoided

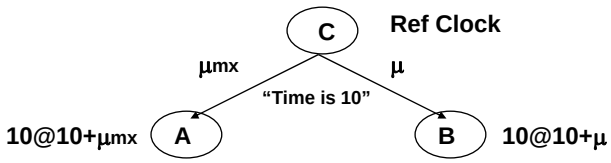
Computing Clock Adjustments

T16

- Periodically, clocks adjust their readings by some 'right' amount. We will see three approaches to obtain this adjustment amount

Central Spray:

- A reference clock periodically transmits its value
- it may be one designated clock in the system or an external reference clock (radio time receivers)



- upon receiving the timestamped message, A and B set their readings to the indicated value

μ_{mx} is the maximum time elapsed between timestamping and receiving

so, the initial error can be ($\mu_{mx} - \mu$) which is the maximum variabilities in delays for message processing, queueing and transmission; $\varepsilon > (\mu_{mx} - \mu)$.

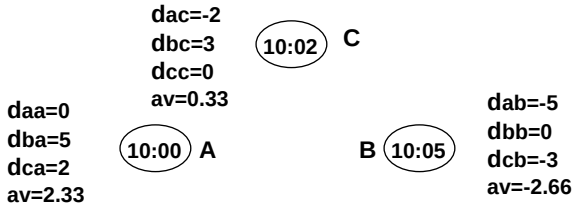
Distributed Approach

T17

A ref clock is a central point of failure

In Distributed approach,

- **each clock reads every other clock**
- **computes relative difference of every clock with itself**
- **use the average of these differences as the adjustment amount. Consider a three-clock system and zero message commn delay**



- **a remote clock has to be read only by message exchange; so, $\varepsilon > (\mu_{mx} - \mu)$.**
- **to avoid network congestion, each clock computes the differences at different (skewed) times, e.g: A reads B and C during [9:45 - 9:50]; B reads C and A during [9:50 - 9:55]; C reads A and B during [9:55 - 10:00];**

Distributed Approach Contd.

T18

Effect of failures

- **No problem if a node crashes before the computation begins; say, node of B crashing before 9:45**
- **Hell of a problem if a node crashes during the computation. Say,**

  node of B crashes at 9:50 after A has read B and Before C could read B.

  It has the effect B telling its value to A but not to C - called two-facing failures - a type very difficult to handle in DS.

  As a result of B's two-facing, C computes an average of -1 and resets its time to 10:01, while A sets its value to 10:02.33. An initial difference of 1.33 even when messages are transmitted in zero time.

Hardware Approach:

- **Use special hardware circuits to**

  (i) timestamp the clock-synch messages just before their transmission onto the physical medium (at the link layer level.)

  (ii) note the arrival time and copy the timestamp of an incoming clock-synch message soon after it is received from the medium.

- **$(\mu_{mx} - \mu)$ is now only the maximum variabilities in message transmission delays**
- **can achieve ε in pico seconds**

Use of Total Order

T19

- **Using total order to construct distributed algorithms**
  - **processes cooperate by exchanging messages; there is no central synchronising process or shared storage.**
- **Example: exclusive use of a single resource (mutual exclusion)**
  - **consider a system composed of a fixed number of processes which share a single resource. Only one process can use the resource at a time, so processes must synchronise themselves to avoid conflicts.**
  - **An algorithm is required for resource sharing under the following three conditions:**

  (i) a process that has been granted the resource must release it before it can be granted to another process;

  (ii) different requests for the resource must be granted in the order in which they were made;

  (iii) if every process that is granted the resource eventually releases it, then every request is eventually granted.

  Note: using a central resource server process for serving requests in the order they were received by the server is not a correct solution (there is no guarantee that the server will receive the messages in increasing timestamp order)

T20

- **Assumptions :**

  (i) Events can be ordered according to the total order relation $\Rightarrow$.

  (ii) For any two processes P_i and P_j, messages sent by Pi to Pj are received in the sent order.

  (iii) A process can send a message directly to every other process.

  (iv) Each process maintains a request queue for its own use (the queue is not accessible to other processes)

  (v) Initially: Po has the resource and all the request queues have a single message; 'T_0: P_0 requests resource' and T_0 is less than the clock value of any clock of a process.
- **The algorithm is defined by the following five rules:**

  (i) To request a resource, P_i sends the message 'T_m: P_i requests resource' to every other process and also puts this message in its own request queue. (note: T_m is the timestamp of the msg)

  (ii) When a process P_j receives the message 'T_m: P_i requests resource', it places it on its request queue and sends a timestamped acknowledgement message to P_i. (note: the following optimisation is possible: the ack message need not be sent if P_j has already sent a message to P_i with a timestamp greater than T_m.

T21

(iii) To release the resource, the process using the resource (say P_i) removes any 'T_m: P_i requests resource' message from its request queue and sends 'T_n: P_i releases resource' message to every other process.

(iv) When a process P_j receives a 'T_n: P_i releases resource' message, it removes any 'T_m: P_i requests resource' message from its request queue.

(v) A process (say P_i) is granted the resource when the following two conditions are satisfied:

(a) there is a 'T_m: P_i requests resource' message in its request queue which is ordered before any other request in its queue; and

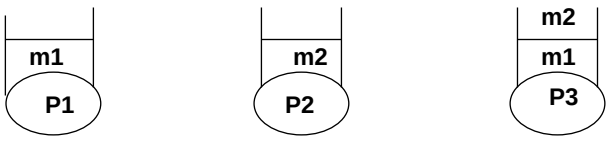
(b) P_i has received a message with timestamp greater than T_m from every other process.

(notes: (i) both the above conditions are tested locally.

(ii) condition (b) above ensures that Pi "knows" that there is no request message that comes before its own request.

T22

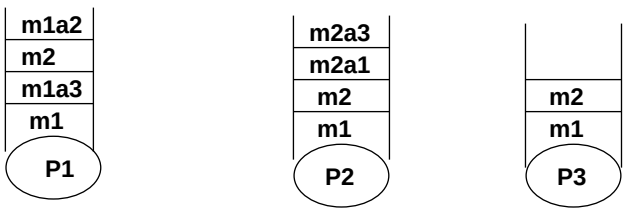
- **Example:**



m1: '6:P1 requests resource', m2: '7:P2 requests resource'

(i) P1 and P2 make requests; m1 gets smaller timestamp than m2; P1 and P2 have not yet seen each other's request.

(ii) P1 (fig. below) can use the resource as its request is ordered before all other requests.



miaj: ack of req msg mi by Pj

Coursework 1: Building a Total Order service

T23

- **This service satisfies the following three conditions in delivering the received messages to processes:**

  (C1) Say messages m and m' are sent to process p. If sending of m happened before sending of m', then the delivery of m to p must happen before the delivery of m' to p.
  (respects "happened before")

  (C2) Say messages m and m' are sent to processes p and q. The delivery of m to p happens before the delivery of m' to p if and only if the delivery of m to q happens before the delivery of m' to q (provides identical order)

  (C3) any message sent to process p is eventually delivered to p. (liveness)

Assumptions

T24

- **(A1) Processes have no access to synchronised physical clocks;**
- **(A2) The communication subsystem guarantees that a sent message is eventually delivered to the destination, but no bound on transmission delays;**
- **(A3) For any two processes p and q, messages sent by p to q are received in the sent order;**
- **(A4) Sending of a message to one or more destinations is a single event within the sending process and,**
- **(A5) Processes never crash and are uniquely ordered; this ordering is known to each process.**

Required:

- **A distributed design for the total order service.**

Coursework 2

T25

A Bank has implemented a computerised fire detection system. All the fire detectors within the Bank are connected to a computer (Computer A). Upon receiving a fire signal, computer A will send one or more "fire!" messages to the computer at the local fire station (Computer C), and also to another computer in the Bank (Computer B) that will activate water sprinklers. Computer B is connected to sensors that inform B only if the fire is put out. If the fire is put out then B will deactivate sprinklers and send a "fire out!" message to computer C. Fire-detectors, sprinklers, sensors and computers within the Bank are adequately fire-proofed and can be regarded to be reliable.

Any message sent is received within a maximum period of *d* time units and *d* is known.

Also known is *e* - the maximum time that can elapse between Computer B activating sprinklers and the sensors informing B if the fire has indeed been put out.

Coursework 2 Continued

T26

The fire station will send the fire engines to the Bank only if no “fire out!” message comes within (2*d*+*e*) time after a “fire!” message has been received.

Figure (a) shows a case which does not require fire engines to be sent.

Figures (b) and (c) show cases where fire engines need to be sent.

In Figure (b), the fire is not put out by the water sprinklers,

In Figure (c) a second fire starts after the first one has been put out and it cannot be put out by the water sprinklers.

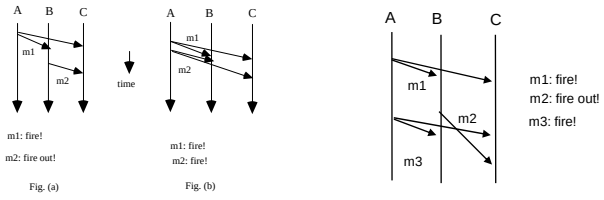


fig. (a) **fig. (b)** **fig. (c)**

Coursework 2 - Questions

T27

- The system is said to be designed correctly if it is guaranteed that fire engines are sent to the Bank whenever a fire cannot be put out by the sprinklers, and not in any other circumstances. Argue that the system design is flawed.
- Produce a correct system design by assuming that A and B use logical clocks and timestamp their messages as per their logical clocks.
- Discuss the conditions under which your answer to ii) will be correct if A and B use physical clocks instead of logical clocks. Assume that the physical clocks are synchronized within a known bound ε.

ATOMIC TRANSACTIONS (ATOMIC ACTIONS)

AT1

- **Atomic transactions are used for controlling operations on persistent (long-lived, disk based) shared information:**
  - **files, databases, and in object oriented systems, objects (e.g., instances of C++ classes)**
  - **we will use the term object to mean any such entity**
- **Operations upon remote objects are invoked through the use of remote procedure calls**
- **Atomic actions then ensure that only consistent state changes to objects take place despite concurrent access and any failures.**
  - **either all the intended changes to the objects take place, or none take place**
  - **in other words, transactions give atomic 'all or nothing' property.**
- **Topics to be covered:**
  - **'ACID' properties**
  - **Transaction primitives**
  - **Concurrency control: serializability and two phase locking**
  - **Two phase commit protocol: the coordination protocol between client and servers.**

AT2

- **A transaction has ACID properties:**
  - **Atomicity: the all or nothing property which ensures automatic recovery and restoration of prior consistent state in the event of failures which would result in the computation terminating in an inconsistent state.**
  - **Consistency: the serializability property which ensures that transactions are scheduled to run such that they execute without interference (equivalent to some serial order execution)**
  - **Independence: the 'containment' property which ensures that the updates performed by a transaction become 'visible' to other transactions only after it terminates.**
  - **Durability: the persistence results property which ensures that results produced by a successfully terminated transaction are not destroyed by subsequent failures.**

(note: in books, you will find slightly different explanations of ACID, as there are many ways of stating transaction properties).

AT3

- **Transaction primitives (operations): Programming using transactions requires the use of special primitives, to be provided by the transaction support system:**
  - **begin-transaction (equivalently, begin-action, or some such operation):**
    - **the command indicating the start of the transaction**
  - **end-transaction (equivalently, end-action):**
    - **A 'commit' protocol is executed to terminate the transaction. This has one of two possible outcomes:**
      - **normal termination (transaction commits): normal ending of the transaction, all updates to data are migrated to disk to make results durable (also called commit the results).**
      - **aborted termination (transaction aborts): transaction terminates without producing any results. This would be the case if due to failures, normal ending of the transaction is not possible.**
  - **abort: explicitly aborting the transaction under the control of the program.**

AT4

- **Example programming interface:**
- **suppose, RPC is available to clients via a 'call' primitive; let the application objects be at two remote servers, S1 and S2.**

```
begin-transaction (..)
   call (S1, lock,....)   // invoke a lock operation for locking objects at S1
   call (S1, OP1,.....)  // invoke operation OP1, eg, debit account
    .......
   call (S2, OP2,.....)  // invoke operation OP2, eg, credit account
    .......
   if some condition then abort   /* changes at S1 and S2 undone
   else
end-transaction (..)   // commit protocol tries to make changes performed
                          at S1 and S2 durable, if this is not possible, transaction
                          is aborted.
```

- **Note: modern transaction systems, like Arjuna, hide the details of RPCs and servers inside stubs, so the programming interface is much simpler.**

AT5

- Simplified Arjuna code. Assume you want to manipulate an object of class Exampleobj, whose state (named uid) is stored on disk. The program is passed through the C++ stub generator to produce client and server stubs for B.

```
1.  Exampleobj B (..uid...);
2.  AtomicAction A;
3.  A.Begin();
4.  B.op();
5.  if (...) A.Abort();
6.  else A.End();
```

1: constructor stub of client creates a server process; this process loads the server program with class code for Exampleobj, linked to server stub. The uid (unique identifier) and related information indicates the file where the state of the object is stored.
2: An instance of AtomicAction class is created.
3: start of action
4: The method of Op contains the locking operation.

AT6

- Overview of approaches used for supporting transactions.
- Failure assumptions:
  - Communication, node failures: same as in RPC discussions. Plus: a node failure does not destroy data stored on disk.

(i) Concurrency control:

- (a) concurrency control operations (eg. read, write lock) are used for accessing shared objects.
- (b) While the transaction is running, the locks are held, to prevent other transactions from accessing the objects.

(ii) End-transaction:

- (a) locks are released during the commit protocol, so changes made to objects become 'visible' only after the transaction ends.
- (b) All the updates are forced to the disk, so subsequent node crashes do not destroy the results. If required, updates can be forced on to n+1 disks, to cope with n disk crashes.

(iii) client, server crashes during the execution of the transaction lead to transaction aborting.

AT7

- **(iii) gives atomicity, as no partial results are produced.**
- **(i,a) gives consistency, as conc. control prevents interference between transactions**
- **(i,b) and (ii,a) give independence.**
- **(ii,b) gives durability.**

AT8

- **Understanding Serializability**

Consider the following example:

x = 5, y = 2

P1	**1: x := x + y**	**1: y := x - y**
	2: z := x	**2: q := y**

P2

Suppose we run P1 and P2 sequentially, one after the other:

S1: P1 ; P2 (P1 followed by P2) will produce x= z=7, y=q=5
S2: P2 ; P1 (P2 followed by P1) will produce x=z=8, y=q=3
Both S1 and S2 are serial schedules (as programs are run one at a time; there is no interference between programs).

Suppose now P1 and P2 are run concurrently. There are several possible ways the program steps get executed.

AT9

- **Here are three execution sequences for P1 || P2 (we use || to denote concurrent execution):**

A. {P1(1) || P2(1)} ; {P1(2) || P2(2)} will produce x=z=7, y=q=3

B. {P1(1) ; P2(1)} ; {P1(2) || P2(2)} will produce results same as S1

C. {P2(1) ; P1(1)} ; {P1(2) || P2(2)} will produce results same as S2

- **We call B and C as serializable schedules as they produce results that are same as some serial execution of programs.**
- **A on the other hand is not a serializable schedule, as no serial execution will produce those results (programs P1 and P2 interfere with each other).**
- **Transaction executions are required to be serializable (to guarantee that transactions do not interfere with each other).**
- **So, what is required is a concurrency control technique that permits only serializable schedules for concurrently running transactions**

AT10

- **Concurrency control using locking.**
  - **Lock an object (file, database record, ...) before using it. (*Release the lock after using the locked object.*)**
  - **read lock: reading can be shared (multiple readers are allowed). A read lock is permitted provided the object is not write locked.**
  - **write lock: writing is exclusive; a write lock is permitted only if there is no read or write lock applied on the object.**
- **Lock conflict:**
  - **Trans. A wants a write lock on object f1, trans. B has a write lock on f1.**

    this is called a write - write conflict.

    Similarly, there can be write - read conflict, or read - write conflict.

    response of A: either wait for the lock, abort itself or force B to abort (an aborting transaction releases held locks).

    Note: there is no read - read conflict.
- **Deadlocks:**
  - **If a transaction's response to a conflict is to wait for the lock, then there is the danger of a deadlock occurring.**

AT11

T1 has locked f1 and wants to lock f2, creating a conflict
T2 has locked f2 and wants to lock f1, creating a conflict.

T1 --> <-- T2

T1 'waits for' T2 and T2 'waits for' T1. In general, the 'wait for' cycle could be arbitrarily large.

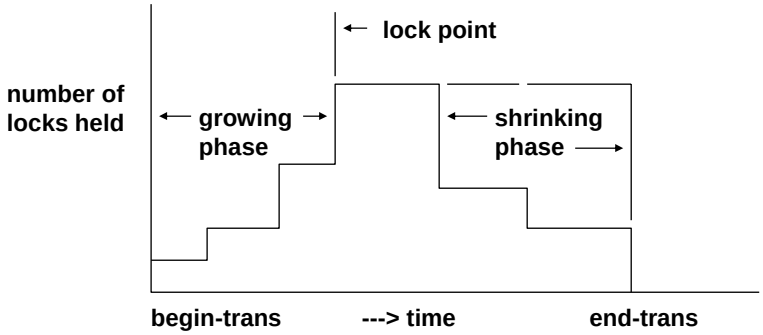
Deadlock Treatment:

Prevention: Use techniques that prevent formation of 'wait for' cycles.
- A transaction waits only if there is no danger of a deadlock
  This is usually simple to achieve.

Detection and recovery: Deadlocks are allowed to occur. Techniques are used for detecting 'wait for' cycles, and then breaking the cycles by aborting relevant transactions.
- detection of 'wait for' cycles in a distributed system is not simple, particularly in the presence of failures (lost messages, node crashes). Also, the necessary information is distributed, not available in a single place.

AT12

- **Two Phase Locking:**



Transactions use the two-phase locking rules:
(i) A transaction must obtain a lock on an object before using it;
(ii) growing phase: locks are acquired and acquired locks are not released;
(iii) shrinking phase: acquired locks are released and no new locks are acquired.
Theorem: Two-phase locking permits only serializable schedules.
Proof: not done here.

AT13

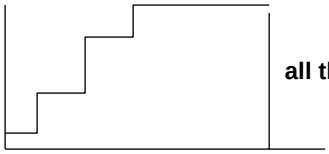
A transaction at some point in time must hold all the locks on objects it is using in its computation. This point is called the lock point.

Note: A transaction can (and does) do processing on the objects it has locked during the growing phase. That is, it is not necessary to obtain all the locks first (reach the lock point) before processing.

Shrinking Phase:

- **Suppose T1 updates f1 and then the lock on f1 is released; later T1 is aborted. Assume in the mean time, some transaction T2 puts a read lock on f1; so T2 will be reading 'dirty data'. So, T2 will have to be aborted as well.**
- **To avoid this 'cascade abort', shrinking phase is made 'instantaneous' by releasing all the locks together, as indicated below. This ensures the independence property, a transaction can abort without causing cascade aborts.**

Strict Two Phase Locking



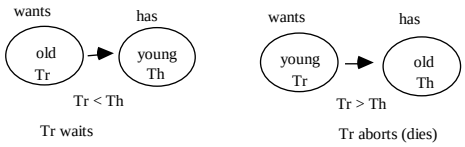
all the locks released together

AT14

- **Deadlock Prevention**
  Use techniques that prevent formation of 'wait for' cycles.
  - **Simple Approach: Transaction never waits. If the transaction requesting a lock cannot obtain the lock (due to conflict), then the requesting transaction aborts.**
    - **This is a simple and effective approach if conflicts are rare (so aborts are rare).**
      - **Used in the Arjuna system.**
  - **A more careful approach is to permit waiting only if there is no danger of a deadlock.**
    - **Transactions are assigned numbers (timestamps) using logical or physical clocks, such that transactions numbers are totally ordered.**
    - **Transaction timestamps are used for conflict resolution (deciding whether to wait or not).**

AT15

(a) WAIT-DIE SYSTEM



Let Tr denote the transaction requesting a lock and Th denote the holder of the lock, with whom the conflict occurs.
Then the conflict resolution rule is:

<u>if</u> timestamp of Tr< timestamp of Th <u>then</u> Tr waits <u>else</u> Tr aborts

So, only old transactions wait for young

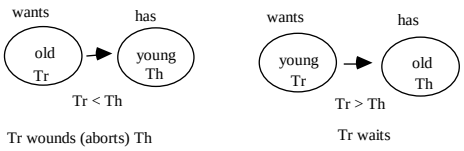
AT16

(b) WOUND-WAIT SYSTEM

Here the conflict resolution rule is:

<u>if</u> timestamp of Tr > timestamp of Th <u>then</u> Tr waits <u>else</u> Tr forces Th to abort.

Here, only young waits for old; so an old transaction succeeds, forcing its way to completion.



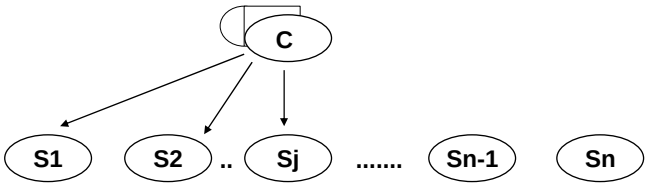
AT17

- **Implementing Transactions:**
- **Strict two phase locking ensures C, I of ACID.**
- **A and D (Atomicity and Durability):**

  Objects keep their states on crash proof storage (also called stable storage).
  - **Such a store can be obtained by storing data on n+1 disks, that will tolerate up to n disk crashes.**
  - **Clients and servers make changes on copies of objects (rather than on original objects).**
  - **If the transaction is to be aborted, these copies are discarded, so original objects remain unaffected. (this is also called rolling back the transaction).**
  - **If the transaction is to be committed, copies are first forced on the stable store, and once this has taken place, then the copies are treated as the new version of the objects (changes are committed). Old versions can be discarded.**
  - **A two-phase commit protocol (two message rounds protocol) is used to ensure that client and servers forming a transaction together either abort or commit.**

AT18

- **Understanding the need for a commit protocol**



Suppose a transaction involves a client process (C) and servers S1, S2,...,Sn. Client starts sending 'commit' message to the servers, but crashes halfway. Assume S1,..Sj get the commit message, so commit the changes. Servers Sj+1,..Sn find that client has crashed, so they abort.
This is not 'all or nothing' behaviour. What is required is that C, S1,....Sn together either commit or abort.
This requires a multiphase protocol (minimum of two rounds).

AT19

- **Two-phase commit protocol**
  - **The termination of the transaction is carried out under the control of the client (called the coordinator). The first phase is used by the coordinator to determine the outcome for the transaction (commit or abort); the second phase is used to enforce the decision taken by the coordinator.**
  - **Every node maintains a file called the 'transaction log' or 'intentions list' on stable store. The log maintains information about the transactions that node is taking part. Special techniques are used to ensure that writing to the log is atomic (all the information is written or if a failure occurs during writing, then incompletely written information is discarded during crash recovery).**
  - **Every node has a recovery manager process that is executed during crash recovery of the node. The recovery manager process scans the transaction log and tries to terminate the transactions that were interrupted by the crash.**

AT20

- **Algorithm:**

Coordinator (Client)	Server
Phase1: ***send 'get ready' to all servers*** ***IF all reply 'yes' THEN {verdict := commit; write verdict, transaction identifier, server node addresses on the transaction log}*** ***ELSE verdict := abort;***	**Phase1:** ***wait for 'get ready' command;*** ***IF none coming THEN abort;*** ***command received and server wants to commit THEN {copy new version of objects on the stable store; write transaction identifier, coordinator node address on the transaction log; send 'yes' to the coordinator} ELSE send 'no'***
Phase2: ***send 'verdict' to all the servers***	**Phase 2:** ***wait for the 'verdict'; carry out the command;*** ***IF no verdict coming THEN keep on checking with the recovery manager of the coordinator for the decision.***

AT21

- **Remarks:**
  1. **In phase one, a server can say 'no'; a 'no' acts as a veto (transaction must abort). If a server does not receive the getready command (within some timeout), it can abort.**
  2. **In phase 1, if a server says 'yes' then it is in a position to either commit or abort as it has both 'old' and 'new' states of objects. Once a server has said 'yes', it must get the decision from the coordinator (a server cannot decide by itself to commit or abort).**
  3. **If the client crashes in phase one, before the log is written, then no information about the transaction survives on that node. This causes the transaction to eventually abort. Also, if verdict is to abort, then the coordinator does not make a log entry. So, no information about a transaction in the log means the transaction is to abort.**
  4. **If a server crashes after making log entry, then upon recovery, its recovery manger uses this log entry to commit or abort by checking with the coordinator (or the recovery manager at the coordinator node)**
  5. **If a server crashes before making a log entry, then no information about the transaction remains in the log, and the transaction eventually must abort.**
  6. **This is a blocking protocol. If the coordinator crashes in phase 2 and remains crashed, then some servers could wait for ever. Unfortunately, no truly non-blocking protocol exists.**

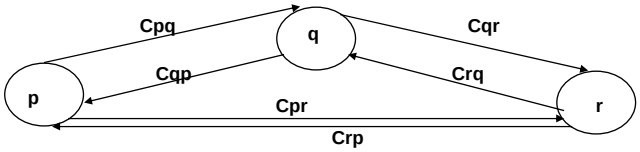
Determining the Global State in DS

gss1

- **Taken from the Article by Chandy and Lamport in ACM Transactions on Computers, Feb 1985, 3(1), pp. 63 - 75.**

A Distributed System D is Modelled as

- **a set of processes {p, q, r,...} connected by unidirectional communication channels**
- **processes need not be fully connected, but be connected to each other**



- **Unidirectional channels model the flow of messages from one process to another**
- **p sent n messages to q**
- **q received only m of them**
- **Cpq is in 'possession' of n-m messages which is the state of the channel Cpq**

Global State and its Determination

gss2

What is a global state of the Distributed System D ?

- **a <u>meaningful</u> aggregation of the states of each process in D and each channel c in D**
- **if the state of p indicates sending of m to q, then either the state of Cpq indicates the possession of m or the state of q indicates m having been received (not both and not neither)**

<u>The problem</u>:

Say process p wants to determine the global state at some point during the computation

The solution if D were a centralised computing system:

- **all processes in D have near instantaneous access to a common clock**
- **p informs all processes to record their local state at some future time**
- **collects the recorded state information of processes and works out the states of channels by accounting for sent but not received messages**

Extending this to a Distributed System has two problems:

- **p cannot know the schedule of every other q; at the time p decided q may not be able to record its state**
- **access to common clock or perfectly synchronised clocks is impossible**

The Problem Definition

gss3

The Problem

- A process p in a Distributed system wants to know the global state and enlists the cooperation of every other process
- We should devise an algorithm by which processes record their own states and the states of the communication channels, so that
  - this recorded information can be put together by p to construct a global system state
- This algorithm should not interfere with the underlying computation nor require the use of a common clock or perfect clock synchronisation

The problem can be likened to

- a group of photographers observing a panoramic and dynamic scene of a sky filled with migrating birds
- the scene is so vast it cannot be captured by a single photographer
- each photographer should take shots which they should be able to later piece together into a picture such that
  - each bird appears only once and no bird is missed out
  - the composed picture is a snapshot of the dynamic scene

Motivation

gss4

Why does a process want to know the global state?

- There is an important class of problems that can be solved if global state can be determined
- Let y be a predicate function defined on a global state S of the system D: i.e. y(S) can be evaluated to true or false.
- y is a stable property of D if "y(S) = true" implies that y(S') is true for all global states S' reachable from S
  - if y is true at one point in a computation, it is true at all later points in the computation; once true, always true.
- many problems in distributed computing can be formulated as a problem of detecting whether a stable property holds

- **Examples of stable property:**
  - The system is deadlocked (deadlock detection)
  - The computation has terminated (termination detection)
  - There is no unserviced request with timestamp smaller than the timestamp of my own request (mutual exclusion problem in the earlier section.)

On the Modelling of DS

gss5

Dist System D : processes + unidirectional channels.

On Channels

- they are error free, deliver messages in the order sent, and can buffer any number of messages
- delay in a channel is arbitrary but finite

Channel State

- the state of a channel is the sequence of messages sent along the channel, excluding those received along the channel.

On Processes

- a process is defined by an initial state and a sequence of events
- an event e in a process p is an action that may change
  - the state of p itself
  - the state of at most one channel c incident on p by either
    - sending a message M along c if c is directed away from p
    - receiving M from c if c is directed towards p

Events

gss6

- An event e is defined by
  - the process p in which it occurs
  - the state of p immediately before e (call it s)
  - the state of p immediately after e (call it s')
  - the channel c (if any) whose state gets altered by e
  - the message M that is sent or received along c
- e is defined by a 5-tuple:
  - <p, s, s', M, c> or <p, s, s', null, null> if the occurrence of e does not affect the state of any channel incident on p.
- 'can occur' condition
  - an event <p, s, s', null, null> can occur in a global state S only if
    (i) the state of process p in S is s
  - an event <p, s, s', M, c> can occur in a global state S only if (i) and
    (ii) if c is a channel directed towards p, then the state of c in S is a sequence of messages with M at its head (because this event causes p to receive M).

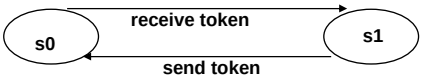
Distributed Computation

gss7

- Let seq = (ei: 0 ≤ i ≤ n)
- seq is a computation iff there exists an initial global state S0, s.t.
  - e0 can occur in S0 and does occur, producing the global state S1,
  - e1 can occur in S1 and does occur, producing the global state S2,
  - e2 can occur in S2 and does occur, producing the global state S3,
  - en can occur in Sn and does occur, producing the global state S(n+1).

An Example;

- the system has two processes p and q with channel c from p to q and c' from q to p.
- the system has one token that is passed from one process to another; a process is in state s0 / s1 when it does not have / does have the token. The state transition diagram:

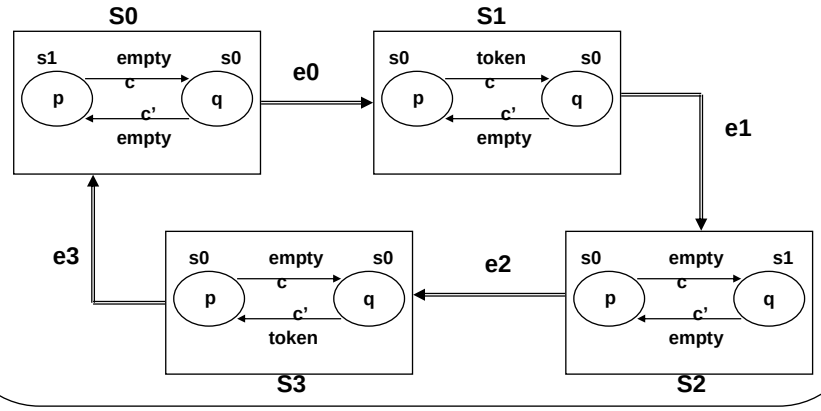


Example Contd.

gss8

- Let seq be: e0: <p, s1, s0, M, c> // p sends token along c
  e1: <q, s0, s1, M, c> // q receives token along c
  e2: <q, s1, s0, M, c'> // q sends token along c'
  e3: <p, s0, s1, M, c'> // p receives token along c'

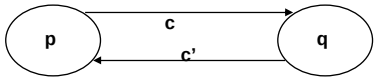
Global states and transitions of the system are:



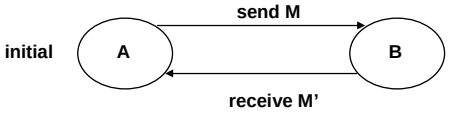
Another (non-deterministic) Example

gss9

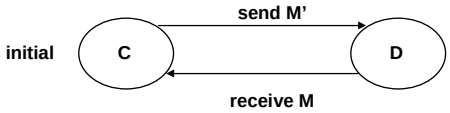
System:



State Transition Diagram for p:



State Transition Diagram for q:



the event sequence for p is:

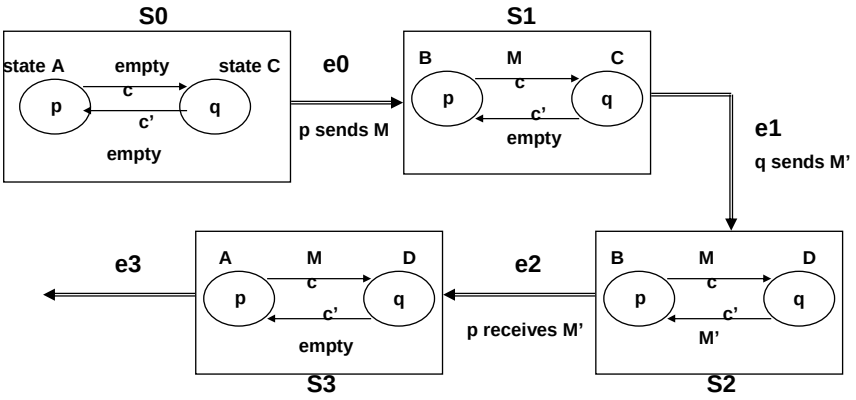
send M, receive M', send M, receive M' ....

the event sequence for q is:

send M', receive M, send M', receive M ....

A computation for the second example

gss10



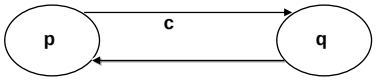
- In the previous (deterministic) example, only one event is possible in every given global state.
- This is not the case here; e.g.

  "q sends M'" could have occurred in S0 instead of e0;, also

  "q receives M" could have occurred in S2 instead of e2

The algorithm - Motivation

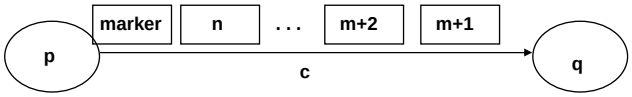
gss11



- Let n be the number of messages sent by p before p's state is recorded
- Let n' be the number of messages received by c from p before c's state is recorded. In any global state n=n'
- Similarly, let m be the number of messages received by q before q's state is recorded
- Let m' be the number of messages delivered by c to q before c's state is recorded. In any global state m=m'
- at any time, n' ≥ m', with (n'-m') being the number of messages in transit along c. This also means that n ≥ m.
- if n' = m' then the state of c is empty,
- else it is the (m'+1)th, ... n'th messages sent by p along c

Motivation Contd..

gss12



We can have q record c's state in the following manner:

- say p sends a marker message after sending its nth message but before sending its (n+1)th message.
- recall, by definition of n and m, p records its state before it sends its (n+1)th message, and q before it receives the (m+1)th message
- make q record the state of c as the sequence of messages it received after it records its own state but before it receives the marker message from p

What if q has not already recorded its state when p's marker arrived?

- q must record its state before receiving (n+1)th message and record the channel state empty

Note: marker messages do not appear in the recorded global state.

- Arrival of the marker message forces q to record its state, if it has not already done so, only before it consumes any new message in c. (This also ensures that n ≥ m.)

Description of the algorithm

gss13

Marker sending rule for a process p.

for each channel c, incident on and directed away from p:

p sends one marker along c after it records its state and before it sends further messages along c

Marker receiving rule for a process q

on receiving a marker along a channel c:

if q has not recorded its state

{ q records its state;
q records the state of c as the empty sequence
}

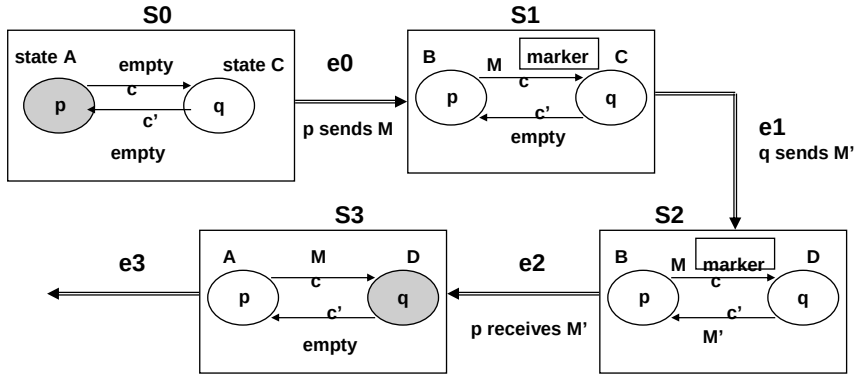
else { q records the state of c as the sequence of messages received along c after q's state was recorded and before q received the marker along c
}

Global-state recording terminates, provided:

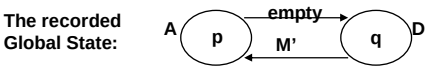
- no marker remains unreceived for ever, and
- a process can record its state in finite time.

The working of the algorithm - example

gss14



- p records its state in global state S0, and sends a marker along c before it sends M; q receives the marker in the global state S3, records its own state as D and the state of c as empty. It then sends a marker along c' which is received by p (in S3 itself). p now records the state of c' as M' - the message it received before the marker from q



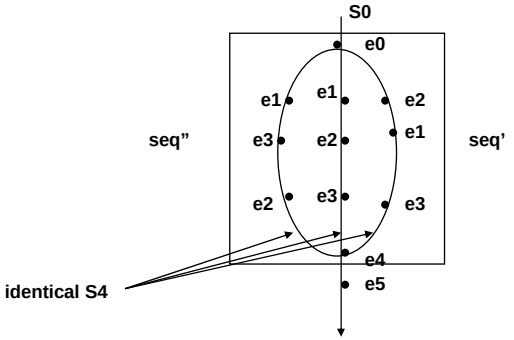
The working of the algorithm - observation

gss15

- Note that the global state recorded by the algorithm is not identical to any of the global states that actually occurred in the computation
- What is the use? To answer that we should understand the property of the algorithm.
- Let S* be the recorded state. Let the seq = (e0, e1, ... ei,.. ef) be the computation in which the algorithm ran. Precisely, recording began just before e0 and ended between (ef-1 ,ef). In the example, ef is e3.
- Sf is the global state when the recording ended; in the example Sf is S3.
- There exists a computation seq' = (e'0, e'1, ..e'j, ..e'f) such that
  - (i) ef = e'f and Sf = S'f
  - (ii) (e'j: 0 ≤ j < f) is a permutation (or a rearrangement) of (ei: 0 ≤ i < f)
  - (iii) for some j, 0 ≤ j ≤ f, S'j is the recorded state S*

Equivalent computations

gss16



- seq' and seq" are permutations of seq.
- all three all have the same final event, occurring at the same state (Sf) and thus produce the same state S(f+1)
- they can be thought of as equivalent ones.
- The property of the algorithm is:
  - if the global state S* recorded during seq is not a global state in seq, then S* will be a global state that occurs during seq or seq"

The seq'

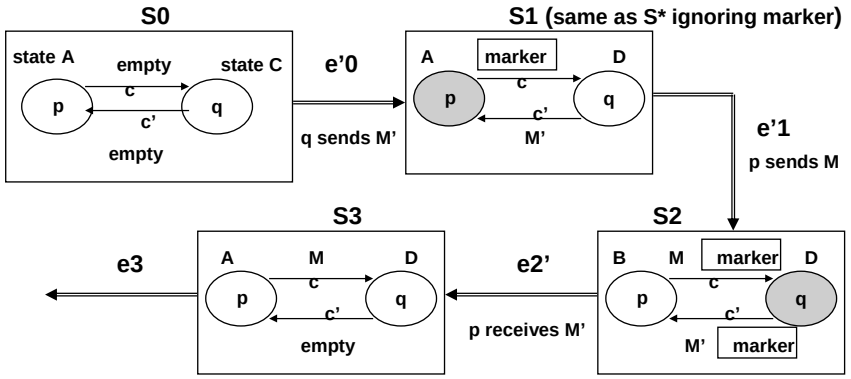
gss17

The seq in the example:

- e0: p (records and) sends M
- e1: q sends M'
- e2: (q records and) p receives M'

The seq' is :

- e0': q sends M'
- e1': p (records and) sends M
- e2': (q records and) p receives M'



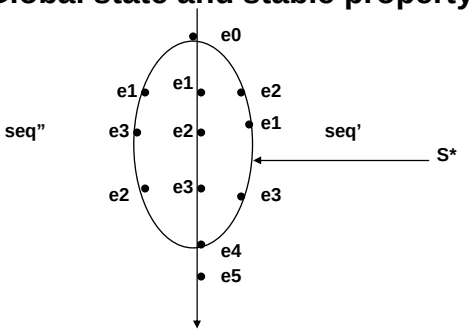
How to determine seq'

gss18

- Every event ei in seq is marked either
  - as a prerecording event if ei occurs in p and before p records its state or
  - as a postrecording event if ei occurs in p and after p records its state.
  - Consider seq in our example:
    - e0: p (records and) sends M -- a postrecording event
    - e1: q sends M' -- a prerecording event
    - e2: p receives M' -- a postrecording event
  - Interchange a prerecording event with a preceding postrecording event
  - here interchange e0 and e1, to get seq'
  - e0 and e1 are necessarily concurrent events, and hence can be interchanged without affecting the end state.

Global state and stable property

gss19

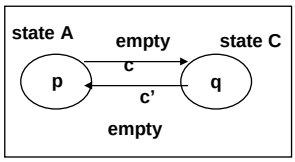


- Let (e0, e1, e2, e3, e4) be the computation during which the global state S* was determined
- Let S* be a global state that occurs in the computation seq'.
- Let a stable property *y* holds before e0 occurs, i.e., at global state S0.
- By definition, once *y* becomes true it remains true in all subsequent states
- Since seq' is an equivalent computation and S* is a global state reachable from S0, *y*(S*) must be true.
- Thus, processes can deduce *y* by determining S* even though S* did not occur in the computation.

An example

gss20

The Initial state S0 is:



- Given that the computation and the execution of the algorithm proceed in the following manner, determine the recorded state:

  e0: p sends M; e1: q sends M';

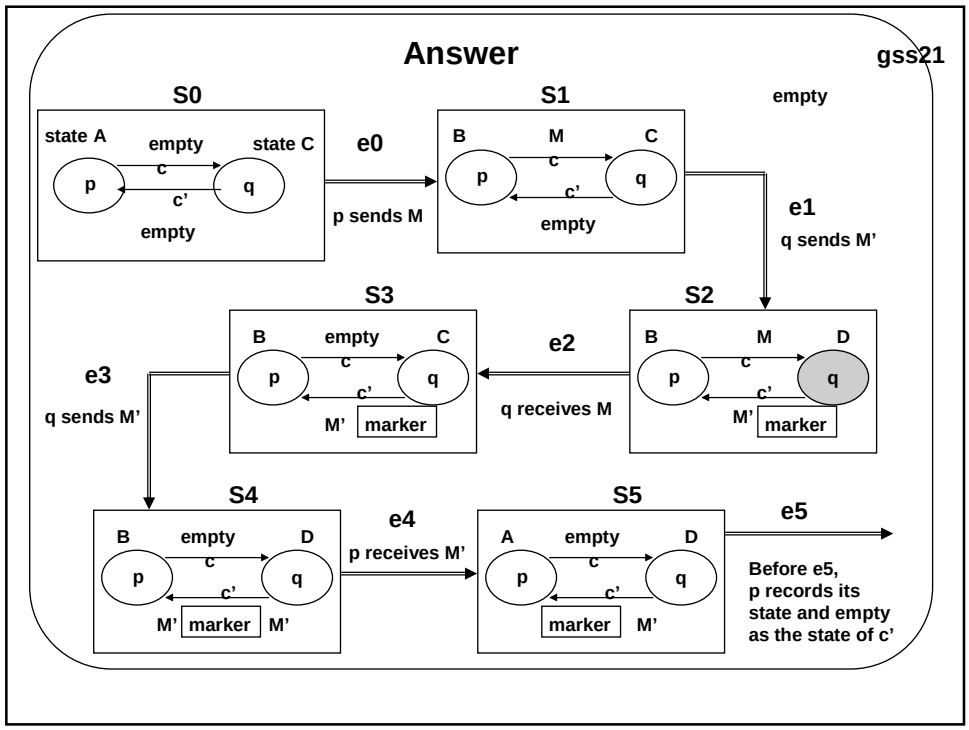
  // q now records its state and sends marker

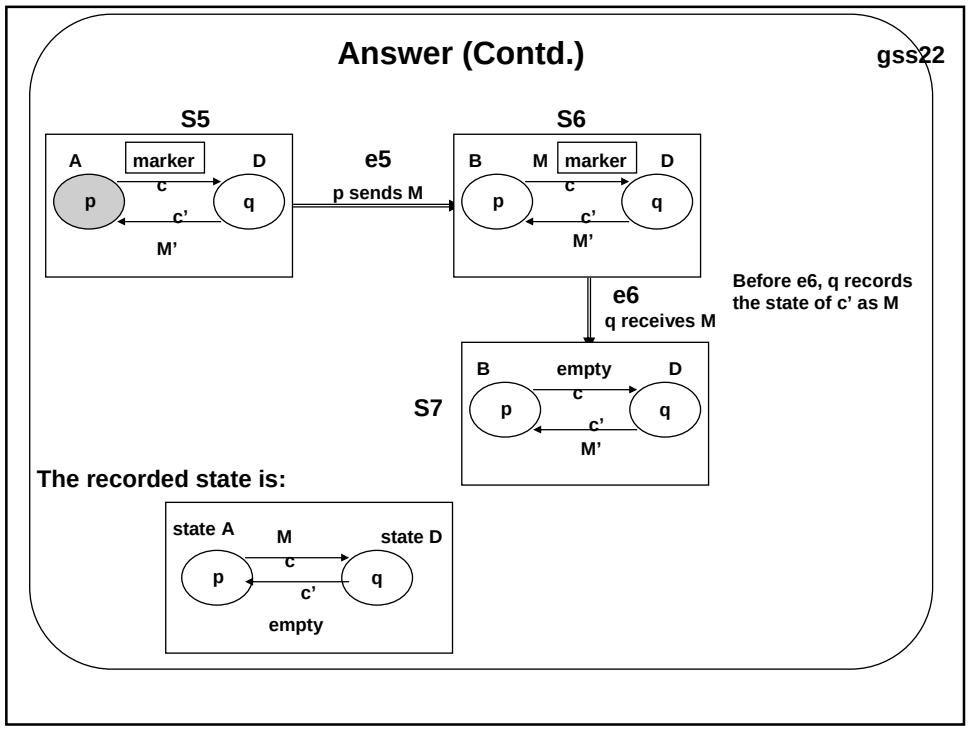
  e2: q receives M; e3: q sends M'; e4: p receives M'

  // p then receives the marker and acts

  e5: p sends M; e6: q receives M.

- Also, find out seq' in which this state occurs.

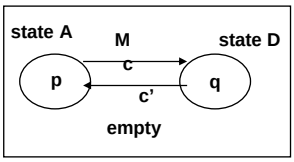

Answer
gss21
S0
state A
empty
state C
c
p
q
c'
empty
e0
p sends M
S1
B
M
C
c
p
q
c'
empty
empty
e1
q sends M'
S2
B
M
D
c
p
q
c'
M'
marker
e2
q receives M
S3
B
empty
C
c
p
q
c'
M'
marker
e3
q sends M'
S4
B
empty
D
c
p
q
c'
M'
marker
M'
e4
p receives M'
S5
A
empty
D
c
p
q
c'
marker
M'
e5
Before e5,
p records its
state and empty
as the state of c'


Answer (Contd.)
gss22
S5
A
marker
D
c
p
q
c'
M'
e5
p sends M
S6
B
M
marker
D
c
p
q
c'
M'
e6
q receives M
Before e6, q records
the state of c' as M
S7
B
empty
D
c
p
q
c'
M'
The recorded state is:
state A
M
state D
c
p
q
c'
empty

The Seq'

gss23

The recorded state is:



To get the seq':

e0: p sends M; (prerecording event)
e1: q sends M'; (prerecording event)
// q now records its state and sends marker
e2: q receives M; (postrecording event)
e3: q sends M'; (postrecording event)
e4: p receives M' (prerecording event)
// p then receives the marker and acts
e5: p sends M (postrecording event)
// q receives p's marker before e6
e6: q receives M ; (postrecording event)

The Rationale Behind Event Shuffling

gss24

- **Note e4 ‖ e2, e4 || e3, but e2 → e3.**
- **So, the orderings <e2, e3, e4>, <e2, e4, e3> and <e4, e2, e3> are logically equivalent.**