ECE 6102 : Dependable Distributed Systems

Fundamentals of Distributed Systems

(A) physical system model

1) a set of nodes
2) a communication network

a node consists of:

1) CPU (1 or more processors)
2) private memory (no shared memory across multiple nodes)
3) non-volatile storage (hard disk or flash memory)
4) network interface
5) clock (independent of clocks on other nodes)

the network can take on many forms but must allow for any node to communicate with any other node in the system (N1)

a closed distributed system is a set of nodes interconnected via a private network used only by those nodes (network structure well defined)

an open distributed system uses a public network, i.e. the Internet, to connect the nodes (network structure is complex, not well defined, but satisfies N1)

B logical system model – a set of concurrently executing processes (or VMs) that cooperate with each other in some fashion

1) a set of processes (or VMs)
2) a set of channels, each pair of processes has a dedicated channel between them (any pair of nodes can communicate)

Notes –

1) # of processes does not necessarily equal # of nodes, multiple processes can execute on 1 node; from now, we will deal solely with processes, not nodes
2) a channel can be either FIFO or non-FIFO; in a FIFO channel connecting nodes x and y, messages are received by y in the same order in which they are sent by x
3) a non-FIFO channel can be made into a FIFO channel by having the sending node attach sequence numbers to each message, the receiving node can then "receive" messages in the order specified by the sequence number (sequence #'s have to be protected – signed or encrypted – if the channel can be attacked)
4) FIFO channel assumption only specifies order of messages received on a single channel but does not address the total ordering of messages received across all channels by a node
5) failures occur to physical or logical components but we are concerned with their impact on the logical system model

attacks can occur both at the physical and at the logical levels

Two Main Categories of Logical System Models

synchronous system model:

all functions are performed within a known bounded time

⇒ processing bounded sequences of code takes bounded time

AND

communication delays across all channels are bounded

asynchronous system model:

no bound can be placed on time to perform any function

⇒ processing time is unbounded

AND

communication delays are unbounded

1) will usually be referred to as "pure asynchronous system"

Notes:

1) in a "pure" asynchronous system, there is absolutely no notion of time so nodes do not even have clocks or they have clocks but there is no bound on their accuracy.

2) "partially asynchronous" or "timed asynchronous" system models assume clocks with given accuracy but unbounded communication delays

1) is not realistic, so we will usually assume 2) when considering "asynchronous systems".

Communication in Distributed Systems

all communication in a distributed system is via message passing (because there is no shared memory)

in message passing, one process sends a message to another process, which receives the message

generic send/receive:

send (p, m) – send message m to process p

receive (p, m) – receive the next message from process p and store in m

receive (*, m) – receive the next message from any process and store in m

two types of message passing –

1) asynchronous
2) synchronous

(different than asynchronous and synchronous system models!)

asynchronous message passing –

sender does not care whether receiver is ready for message or not (since nodes have their own independent clocks, we can not guarantee that a receiving process will reach its receive statement at any particular time with respect to the sender)

sender just ~~puts~~ places message on channel and continues execution (this is called non-blocking execution)

when receiver executes a receive statement, it reads the message off the channel; if no message is present, receiver blocks and waits for message to arrive

note that pure asynchronous message passing requires infinite buffers on channels because a sender can send an arbitrary number of messages on a channel before the receiver receives the first message

in a practical implementation, buffers are finite and sender is forced to block if channel buffer is full (or else some messages might be lost)

synchronous message passing –

each send <u>must</u> block until corresponding receive statement is executed

hence, the send/receive pair form a synchronization point between the two processes

since message is not placed on channel until receiving side is ready for it, the channel does not require buffers

synchronous message passing is sometimes called a <u>rendezvous</u> because whichever statement (send or receive) is reached first waits for its partner

what can be said about the status of a message after a synchronous send statement completes?

in most implementations, all that can be said is that the corresponding receive statement has begun executing, i.e. the send completes when the last bit of the message is placed in the channel but this does not imply that the message has been fully received, i.e. it could be in transit still or only partially received (this relies on network to ensure that message is reliably delivered after it is sent)

some synchronous send/receive implementations have stronger semantics where send return implies message has been fully received – this requires more complex handshaking between sender and receiver

because synchronous communication combines communication and synchronization in 1 primitive, its semantics are easier to reason about

Some applications use asynchronous communication for performance reasons (less idle time spent waiting for other processes)

Higher-Level Communication Operations

can be built out of basic sends and receives

1. broadcast – send(*, m): send m to all processes
2. multicast – send(proc-list, m): send m to all processes on proc-list
3. remote procedure call (RPC):
   used for client-server interaction

client

send(server, request)
receive(server, result)

sometimes combined into 1 function rpc(server, request, reply)
(e.g. MPI's send_recv() function)

server

receive(*, request)
(process request)
send(client, result)

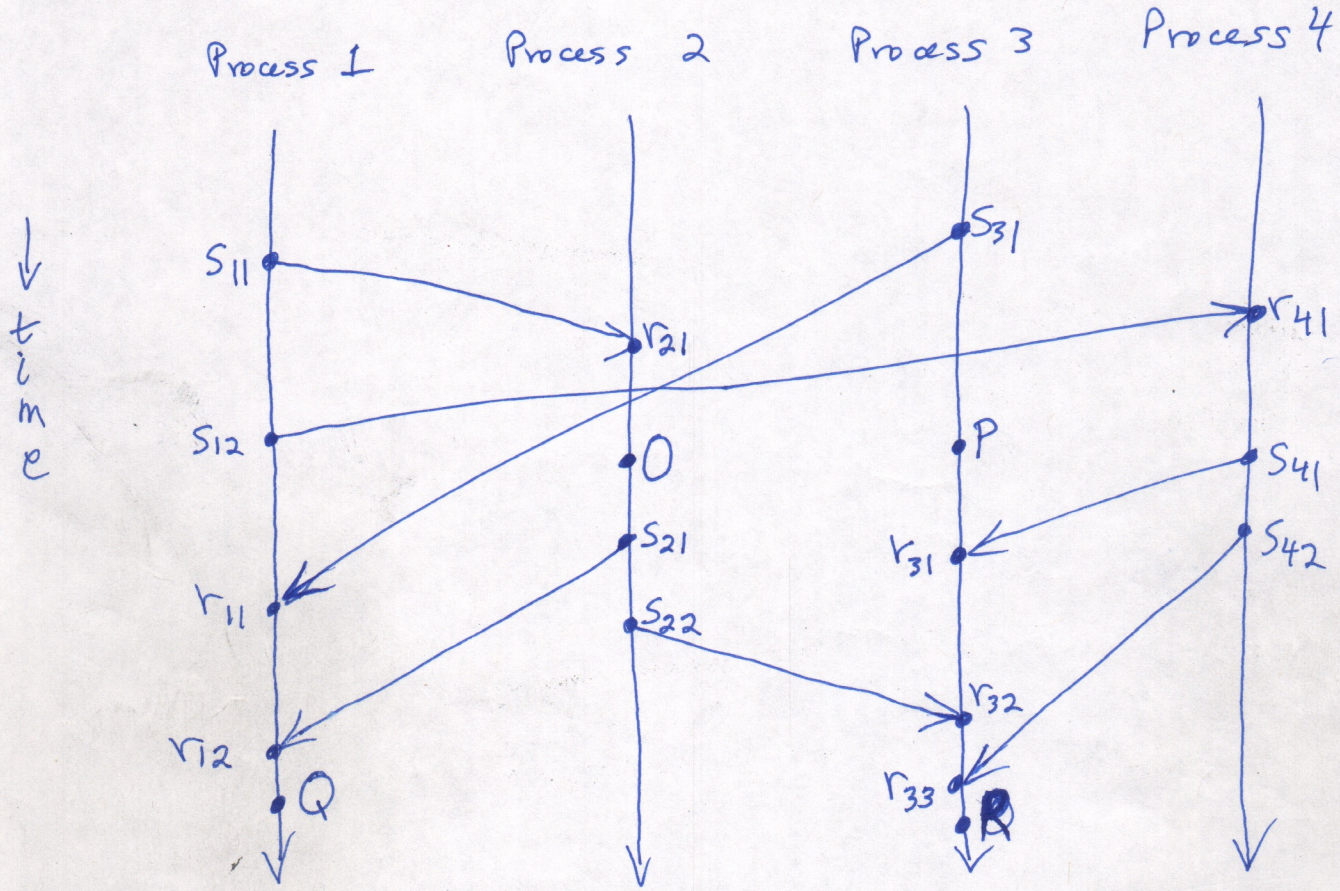
RPC examples:

1) Java RMI (remote method invocation)
2) CORBA remote method calls
3) HTTP get (HTTP put, post, delete are an asynchronous RPC where the requests return a callback ID that allows process to check operation completion status at a later time)

Events and their Ordering

our view of a distributed system – multiple concurrently executing sequential processes that exchange messages (assume synchronous m-p)

pictorial view:



each send or receive is referred to as an <u>event</u>

it is often useful (or necessary) to know in what order things happen in a system

for a sequential process, statements execute in sequential order so this is trivial

in a distributed system, with processes executing concurrently, the order of things is less certain

in fact, in two different executions of the same application with the same inputs, the same operations can be ordered differently depending on process timing

this complicates debugging because you can get bugs that are triggered only for a particular ordering of operations and that particular order might occur only once in a 1,000 executions

what can we say about order in a distributed system?

two operations on a single process can always be ordered, e.g. P "happens before" R in figure

two operations on different processes might or might not be ordered

e.g. O, P can occur in either order

however, O always "happens before" Q even though they are on different processes

this is due to transitivity of "happens before"

O "happens before" S_{21}

S_{21} "happens before" r_{12} (explain) (true for both asynchronous and synchronous message passing) (assumes blocking receives!!!)

r_{12} "happens before" Q

$\therefore$ O "happens before" Q

denote the "happens before" relation by $\rightarrow$

$\rightarrow$ defines a <u>partial order</u> of system events (this order is often referred to as <u>causal order</u>)

partial means some events are ordered with respect to each other and others are not

in other words, there are events a, b such that $a \not\rightarrow b$ and $b \not\rightarrow a$ – such events are said to be <u>concurrent</u> and can happen in either order in 1 execution and in different orders in 2 different executions

e.g. O, P are concurrent events in figure

~~[later we will see an algorithm that uses logical clocks to achieve a total ordering on events in distributed systems]~~ (no longer covered)

Execution Model of a Distributed System

execution is modeled as a sequence of global system states

<u>global state</u> of a distributed system – collection of individual process states <u>and</u> channel states from a single instant of time

not sufficient to have process states because there can be messages in transit (inside channels)

in other words, if p_1 is sending a message m to p_2, then it is possible that the process states are as follows:

p_1 – message m sent

p_2 – message m not received

these states are inconsistent unless the state of the channel connecting p_1, p_2 (which contains m) is included in the global system state

<u>state of a process</u> is the entire contents of the virtual address space of the process (memory contents) plus the internal processor state of the process (program counter plus other internal processor register values)

<u>state of a channel</u> is a list of messages contained in the channel (ordered list if FIFO channel, unordered if non-FIFO) (channel can contain at most 1 message if synchronous message passing is used, can have unlimited number (or up to buffer capacity) for asynchronous m-p)

an <u>event</u> causes the system to move from one state to another

each event is local to a process and affects the states of the local process and at most one of its channels

events are things such as:

- instructions or system calls, which affect only the process state
- send or receive, which affect both process state and the state of one channel

<u>initial state of a distributed system</u> –

1) all processes are in their initial states
2) all channels are empty

<u>possible events</u> – for a given state, all possible events that can occur in that state

<u>next state function</u> – given a state and a possible event, returns the next state of the system

<u>system execution</u> – a sequence of events $E_0, E_1, \ldots, E_n$ such that:

1) $E_i \in$ possible_events(S_i), $\forall i$
2) $S_{i+1} =$ next_state(S_i, E_i), $\forall i$

where S_0 represents the initial state of the system

<u>Example</u> –

<u>P_0</u>

a: x = 7;
b: x = x + 23;
c: send(P_1, x);
d: x = x * 2;

<u>P_1</u>

e: y = 10;
f: receive(P_0, y);
g: y = y + 8;

a possible execution:

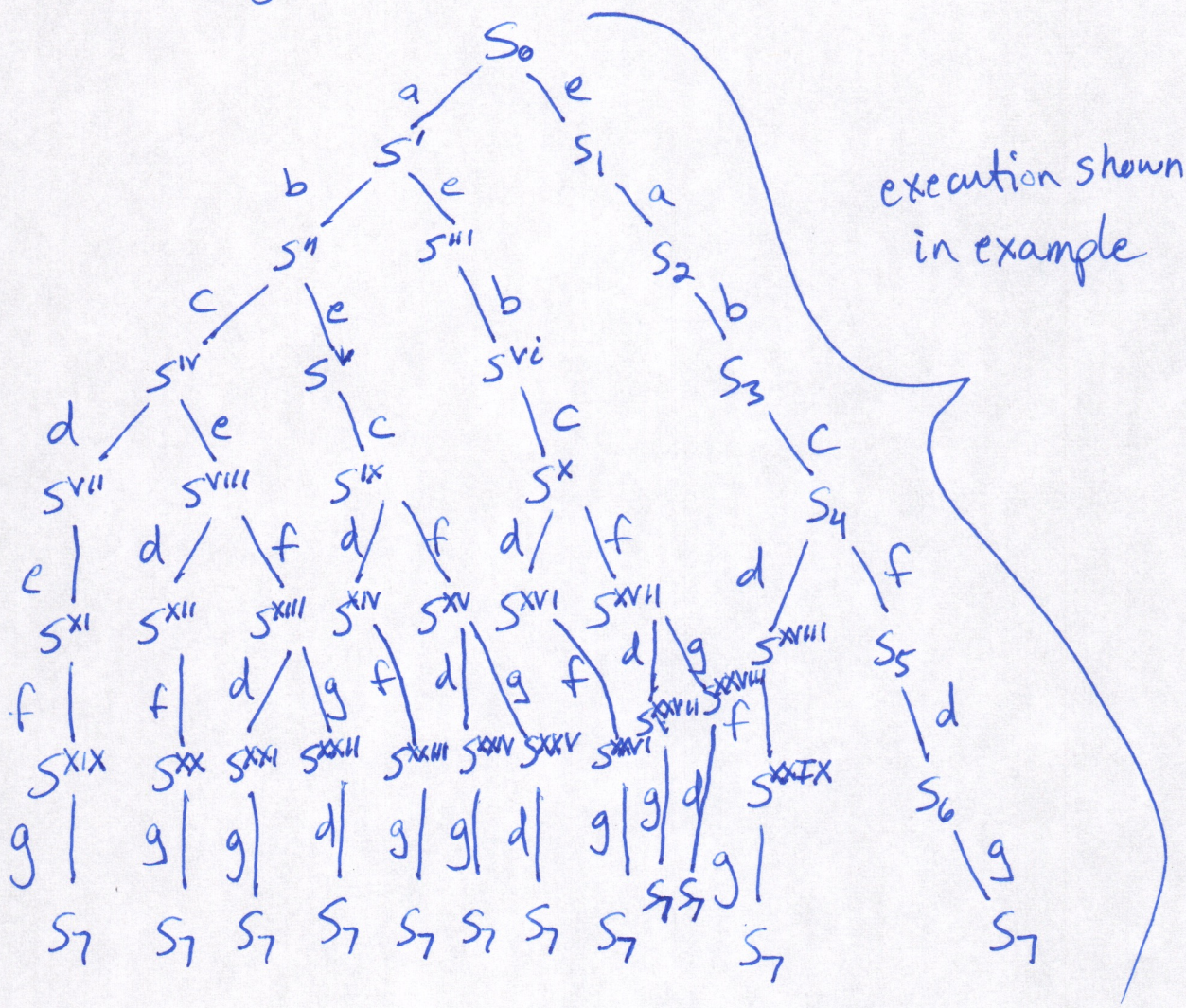
state	possible events	event
$S_0 = x=0, y=0, c_{01}$ empty	a, e	$e = E_0$
$S_1 = x=0, y=10, c_{01}$ empty	a, ~~f~~ → explain why f is not a possible event	$a = E_1$
$S_2 = x=7, y=10, c_{01}$ empty	b	$b = E_2$
$S_3 = x=30, y=10, c_{01}$ empty	c	$c = E_3$
$S_4 = x=30, y=10,$ C_{01} contains 30	d, f (assumes asynchronous message passing)	$f = E_4$
$S_5 = x=30, y=30,$ C_{01} empty	d, g	$d = E_5$
$S_6 = x=60, y=30,$ C_{01} empty	g	$g = E_6$
$S_7 = x=60, y=38,$ C_{01} empty		

note that multiple distinct executions are possible in this system; any time there are multiple possible events, ~~either~~ any possible event could occur each leading to a different next state

multiple possible events in the same state are concurrent events, can be executed in any order, and lead to non-deterministic behavior alluded to earlier

from the initial state of a system, we can construct the <u>reachability tree</u> of the system that contains all possible execution states of the system (some states can occur in multiple locations in the tree)

Reachability tree for example:



this system always ends up in S_7, no matter what order the events happen in

(some of the internal nodes of the tree are duplicate states but are not shown w/ same superscript or subscript, e.g. S_2 is the same as S''')

the reachability tree contains all valid system states, usually referred to as <u>consistent states</u>

any state not in the reachability tree is said to be invalid or <u>inconsistent</u>

example of inconsistent state is a state that shows a message being received before it is sent

from example system: $x=7$, $y=30$, C_0 empty is an inconsistent state

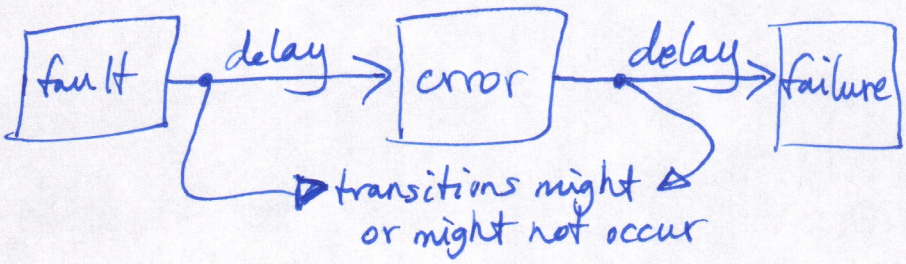
Dependability Fundamentals

<u>dependable system</u> – a system that is capable of correctly performing to its specification in the presence of exceptional events (faults, attacks, natural disasters, etc.)

<u>fault</u> – physical defect or imperfection within a hardware or software component

<u>error</u> – a deviation from correct system state

<u>failure</u> – a deviation from $\underbrace{\text{correct performance}}_{\text{specification}}$ of a system or subsystem



dependability is primarily achieved through redundancy

Where Redundancy is Employed

1) hardware
2) information (coding)
3) time
4) software

Redundancy Techniques

1) replication – redundant identical modules (works only if faults in different modules occur independently)
2) diversity – redundant modules with same function but designed differently ~~(when faults~~ (for design faults and attacks)

Mechanisms to Achieve Dependability

1) fault masking – prevents errors from causing failures, ~~whi~~ provides continuous system operation (sometimes called passive redundancy)

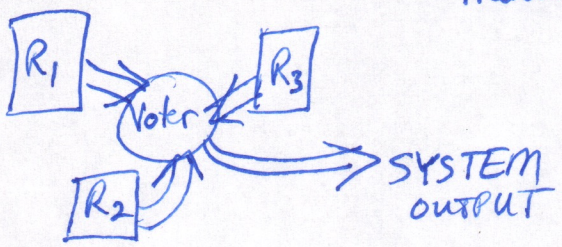
active redundancy

2) a) fault detection
   b) fault diagnosis (optional for transient faults)
   c) recovery/reconfiguration

3) hybrid redundancy – masks faults to provide continuous operation but also performs recovery/reconfiguration to maintain fault tolerance level

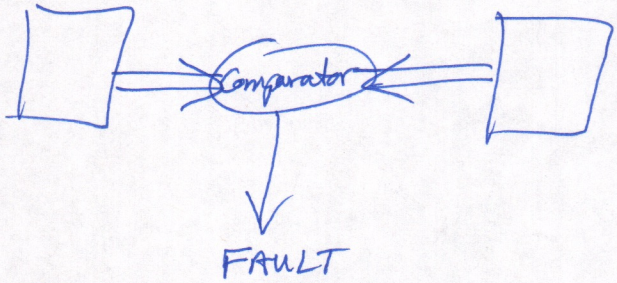
Example of hardware redundancy is modular redundancy, usually triple modular redundancy (TMR) or double modular redundancy (DMR)

TMR:



(fault masking for faults in any single module (replica))

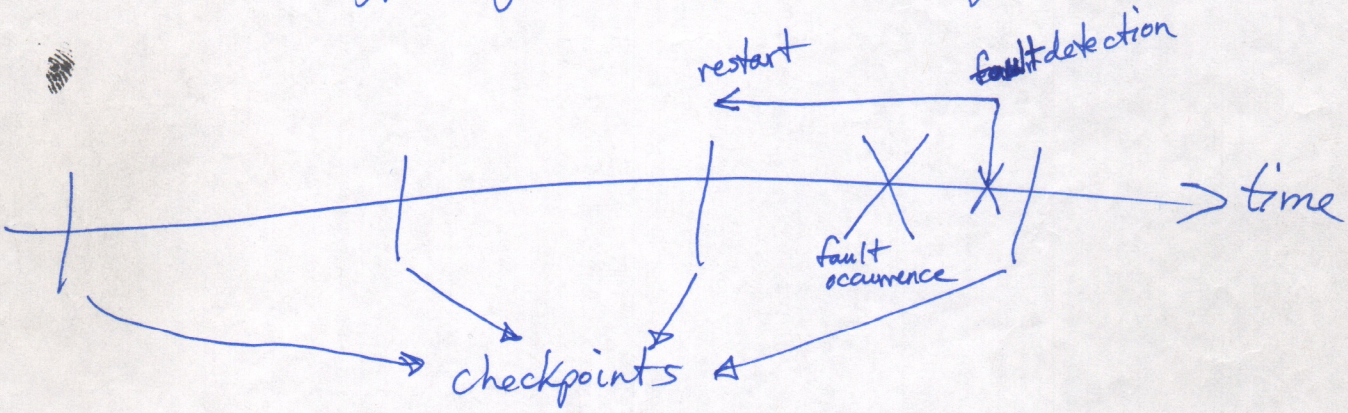
double modular redundancy (DMR)



provides fault detection only, must be combined with diagnosis and recovery mechanisms to achieve fault tolerance

recovery technique – checkpointing with rollback recovery

checkpointing – process of saving the state of a system, typically done periodically



rollback recovery – process of restoring previous state and restarting execution

checkpoint/rollback approaches can tolerate transient/intermittent faults on a single node or any node fault in a distributed system (including a permanent fault)
(explain)

Information redundancy — <u>coding</u>

error-correcting codes, e.g. Hamming codes, are like fault masking

error-detecting codes, e.g. parity or checksum, are like fault detection and have to be combined with retry/recovery techniques

secret sharing, erasure codes — special types of error-correcting codes used in secure storage systems

Time redundancy —

same or modified computation is repeated 2 or more times on same node and results compared, can handle transient/intermittent and some permanent faults

Software redundancy —

1) consistency checks (sanity checks)

2) executable assertions (statements about program's variables that should be true at specified times)

3) n-version programming

4) recovery blocks (alternate algorithms) with acceptance tests

5) general software diversity — compiler-generated diversity, platform diversity, etc.

Faults in Distributed Systems

faults can occur either in network components or in nodes

logical fault models –

1) crash – a fault that causes a component to halt totally (accept no input, generate no output)

2) omission – a fault that causes a component to be selectively non-responsive

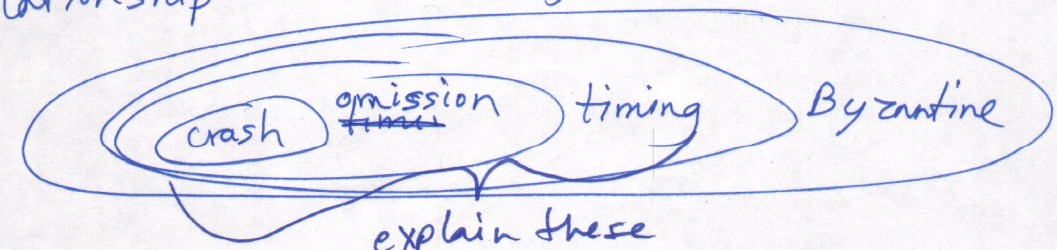
no such fault model in pure asynchronous systems

3) timing – a fault that causes a component to deliver output too early or too late (performance fault)

4) incorrect computation – a fault that causes a component to produce incorrect output in a timely fashion

5) Byzantine – a fault that causes <u>arbitrary</u> incorrect behavior in a component (can be used to model compromised components, i.e. components in control of an attacker)

relationship between logical fault models –



explain these inclusion relationships

incorrect computation Byzantine

faults in network components are typically modeled as either crash, omission, or timing faults

most networks use coding techniques that would allow corrupted messages (incorrect computation faults) to be detected and dropped, hence these faults are transformed to omission or crash faults

for processors, any type of fault can occur

for clocks, common fault models are crash, timing, and Byzantine

runs too fast or too slow – outside manufacturer's specs

fault model for security –
mobile adversary model – attacker takes over machines progressively over time but total # of machines "owned" by attacker at any 1 time is limited

suitable for slower-moving viruses and worms, not for latest ones that propagate extremely fast (Slammer)

Metrics

Reliability – probability that a system operates correctly for some period of time

Availability – probability that a system operates correctly at some instant of time (steady-state availability – fraction of time that system is up on average)

Integrity – probability that system output is correct and system state is uncorrupted

Confidentiality – probability that data is not leaked to anyone not authorized to access it