Components of Dependable Distributed Systems[1]

Tamás Bartha

Technical University of Budapest

Contents:

1. Based on S. Mishra and R. D. Schlichting, "Abstractions for Constructing Dependable Distributed Systems," TR 92-19, University of Arizona, 1992.

System model / 1.

<u>Hardware basis:</u>

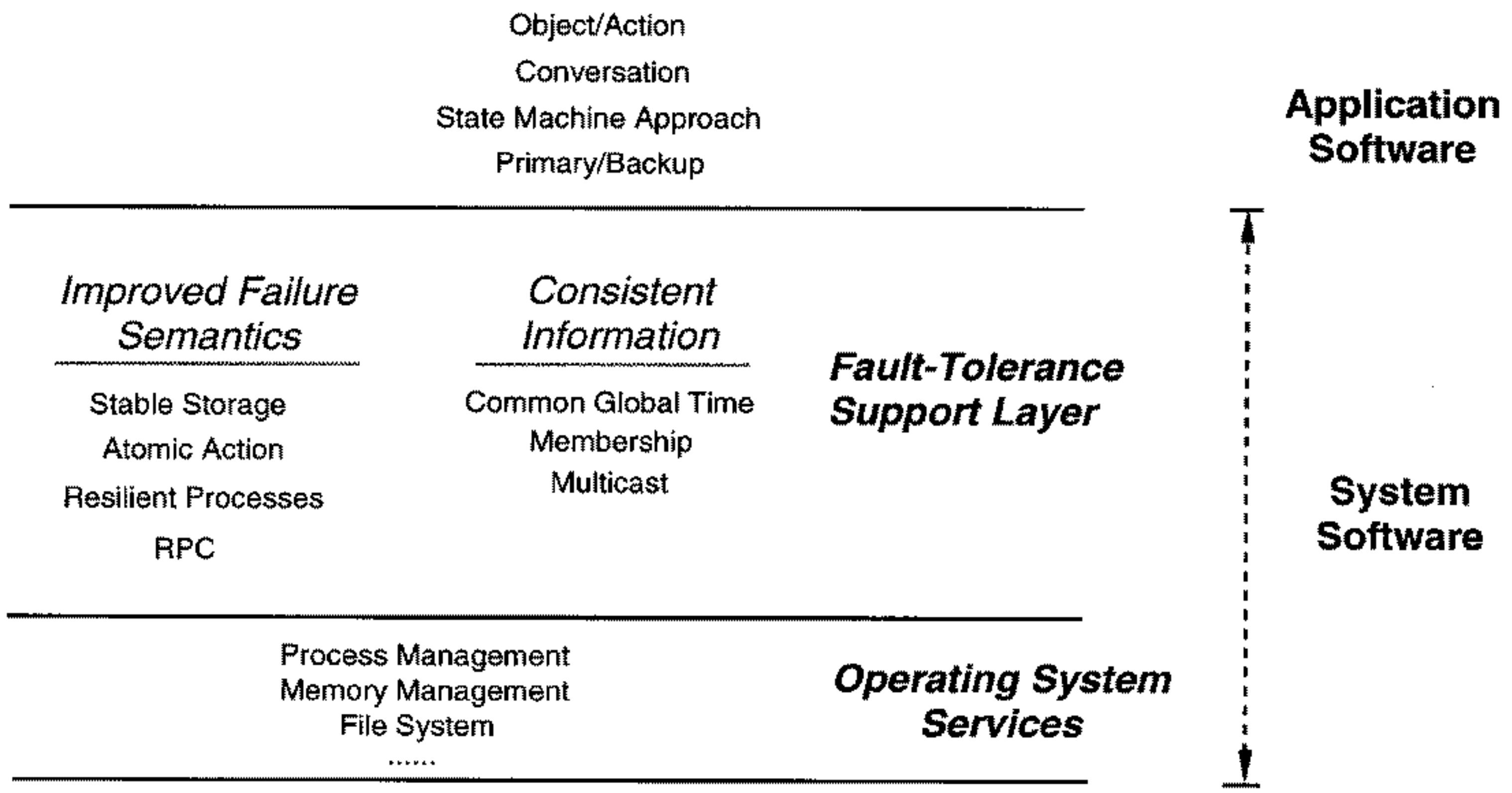
- Collection of processors
  local memory, typically no shared memory
- Communication network
  cooperation based on message exchange

<u>Software structure:</u>

- Application software
- System software
  - standard operating system services
  - fault-tolerance support layer: realizes abstractions
    - standard abstractions with improved failure semantics
    - abstractions providing consistent information

System model / 2.

Overall model of fault-tolerant software:



Implementation of fault-tolerant services / 1.

<u>Programming paradigm:</u>

- **Object / action model**: objects and actions
  - objects: passive; encapsulate a state and operations to change it
  - atomic actions: active; invoke operations to carry out a given task
    - recoverable (either executed completely or not at all)
    - serializable (multiple actions concurrently execute in a serial order)
- **Conversational model**: processes and messages
  - processes periodically *checkpoint* their state onto a stable storage
  - conversation is a structuring technique to assure global consistency
    - recovery line: to avoid the domino effect
    - reliable inter-process communication is required

Implementation of fault-tolerant services / 2.

- **Primary / backup approach**: passive replication
  - services are implemented by multiple processes
    - only the *primary* is active: receives all of the service requests
    - a *backup* process is activated when the primary fails
  - state information is checkpointed or actively propagated to backups
- **State machine approach**: active replication
  - services are implemented as state machines
    - maintain *state variables*, modified by *commands*
    - commands are deterministic and atomic
    - state machines are replicated on a different processors,
  - maintain replica consistency, integrate repaired replicas

Implementation of fault-tolerant services / 3.

Synchrony:

- **Synchronous**: execution time is within a finite and known bound
  - a ~ communication link transmits a data unit in bounded time
  - a ~ processor computes a work unit in a bounded time
  - a ~ network transmits a data unit among nodes in bounded time
- Protocols
  - a *clock-driven* protocol uses clock synchronization methods
    - typically periodic protocols, actions initiated by the common clock
    - assume synchronous processors and communication
  - a *clockless* protocol does not rely on common global time
    - approximate processor and communication synchrony
    - use time-outs and acknowledgments to put a bound on delays

Common global time / 1.

<u>Synchronizing clocks:</u>

- Basic variants
  - internal clock ~ keeps clocks in a certain drift of one another
  - external clock ~ keeps clocks in a certain deviation from a reference
- Properties of synchronized clocks

  a. *monotonicity*: clock is a monotonically increasing discrete counter

  b. *precision*: ~ are within some maximum deviation of each other

  c. *interval preservation*: $)\tau \leq C_i(\quad) - C_t(\ ) \leq (\quad)'$
- Problems
  - random processing and network delay: $\beta > (\quad)(\quad)$
  - n clocks cannot be ~ in the presence of n / 3 Byzantine failures

Common global time / 2.

Algorithms:

Periodically exchange local clock values, then compute a correction factor.

a. **Convergence function** approach:
- correction factor is computed using a fault-tolerant averaging function
- fully connected network, upper bound on transit delay, initial synchronization

b. **Agreement protocol** approach:
- fault-free processors agree on a vector of local clock values
- upper bound on transit delay, maximum number of faulty processors

c. **Synchronizer process** approach:
- periodically elect a synchronizer that collects and distributes local clock values
- upper bound on transit delay, initial synchronization

d. **Probabilistic** approach:
- allows a process to read a clock by a specified precision with high probability
- master-slave arrangement: slaves adjust their clocks to the master clock value

Common global time / 3.

<u>Logical clocks:</u>

- Causality or "happened before" relation: $a \rightarrow b$ if
  - a and b are events in the same process and a occurs before b, or
  - a is the sending of a message and b is the receipt of that message
- Clock value: for any two events a and b if $a \rightarrow b$ then $C(a) < C(b)$
  - logical clock C_i is a counter, incremented between successive events
  - messages contain time-stamps of current local clock values
  - on receipt of a message m_j the process sets $C_i := \max(C_i, C_j) + 1$
  - problem: $a \rightarrow b \Rightarrow C(a) < C(b)$, but $C(a) < C(b) \not\Rightarrow a \rightarrow b$
- Extensions
  a. **version vector**: count of messages received from process P_i
  b. **context graph**: nodes are events, arcs are causality relations

Multicast / 1.

Multicast provides the ability to send identical messages to each process in a group.

Useful in *distributed database update*, *commit protocols*, *managing replicated data*, *distributed synchronization*, and *distributed transaction logging*.

- Properties

  a. **dissemination**: broadcast message to each member of the group

  b. **atomicity**: deliver message to every fault-free process or none

  c. **reliability**: deliver message to failed processes after recovery

  d. order: messages by different processes are in a consistent order

  - **partial / causal order**: concurrent messages are not ordered
  - **semantic dependent order**: depends on the information in message
  - **total order**: messages are sent in the same order to every process
  - **total order preserving causality**: same order, causality relation

  e. **termination**: ~ operation is completed in a known time interval

Multicast / 2.

Algorithms:

1. Using **synchronized clocks**
   - messages include the current local time, order is according to this value
   - atomicity is achieved by treating non-receipt in time $t + \Delta$ as a failure
2. Using **logical clocks**
   - atomicity is achieved using a positive or negative acknowledgment scheme
   - a point-to-point network and crash failure semantics are assumed
3. **Funnel process** approach
   - messages are sent to one process, it assigns a sequence number → order
   - disadvantages: ~ is a single point of failure, performance bottleneck
4. **Byzantine agreement** protocols
   - at least $3t + 1$ processes are required to tolerate t failures
   - recursive algorithms, one less failure in each successive round

Remote procedure call (RPC) / 1.

Extension of the standard procedure call to multiple (client, server) processes.

Clean synchronization semantics, supports network transparency, easy to optimize.

- Properties
  - execution semantics:

    differences from standard procedure calls to cope with uncertainties

execution semantic	**normal termination**	**abnormal termination**
at least once	one or more execution	one, more, or partial execution
exactly once	one execution	one or partial execution
at most once	one execution	atomic execution

  - **orphans**: unneeded server computations after abnormal termination

    server must detect, abort, and eliminate side effects of orphans
  - **call ordering**: invocations and executions must be in the same order

Remote procedure call (RPC) / 2.

Implementation:

- Two distinct components
  - a. name a procedure, locate remote machines, manage message transfer

    at each server a *manager procedure* receives RPC requests

    i. local client: invoke procedure or forward request to another server

    ii. remote manager: accept request and invoke procedure
  - b. detect failures, detect and abort orphans
    - execution deadline, server crash count
    - orphan process adoption by a secondary server
- Replicated RPC
  - support replicated hardware: *client replica set* ↔ *server replica set*
    - generate independent RPC requests to each server in the set
    - concurrent processing; send results back to each client in the set

Membership / 1.

Membership protocols enforce agreement on a unique sequence of member joins and departures, and the way these interleave with regular events.

- Types of membership services
  a. user level: translate failure and recovery into ordered events
  b. system level: *monitor service* to give a consistent view of resources
- Properties
  - **sf-group**: simultaneously failed processes
    - members are removed simultaneously, in the same order
  - **timeliness**: termination is guaranteed within a known time interval
- Network partitions

  Subsets of member processes cannot communicate with each other.
  - clockless ~ protocol: a subset with majority continues to function
  - clock-driven ~ protocol: reconcile divergent views among partitions

Membership / 2.

Failure and recovery detection:

- Failure detection
  - **heartbeat protocol**: processes send periodic *I'm alive* messages
  - puts an upper bound on response time in an asynchronous network
- Notification of recovery: explicitly by the recovering process

Algorithms:

- Clock-driven ~ protocols, broadcast communication network
  a. each process affirms its existence periodically by a *present* message
     - relies on an atomic broadcast service and a message diffusion service

  b. processes include certain membership information in messages
     - communication is controlled by a synchronous TDMA strategy

Membership / 3.

Algorithms:

- Clockless ~ protocols, asynchronous processors
  a. distinct manager process, failure detection
     - the manager process coordinates the local views of other processes
     - two-phase update protocol, three-phase manager selection protocol
     - requires a completely connected network with FIFO channels
  b. distinct manager process, synchrony via message suspension
     - three-phase protocol for the manager, two-phase for the others
     - all the normal traffic is suspended while the protocol is in progress
  c. multiple manager processes
     - uses multicast to agree on the removal of failed processes
     - additional failures during the protocol are handled incrementally

Atomic actions / 1.

A specific computation that is seen as an indivisible state transformation—although it is composed of many primitive steps executed at different times on different processors.

- Serializability / indivisibility

  Concurrently executing a collection of ~ equals to a serial schedule.

  – **two-phase locking**: a lock is associated with each shared resource
    - *growing phase*: lock accumulation; *shrinking phase*: lock release
    - additional locks cannot be acquired in the shrinking phase
    - deadlock detection and elimination schemes must be used

  – **time-stamps**: a system-wide unique number is assigned to actions
    - a shared resource is accessed in their respective time-stamp ordering
    - a request by an action with a smaller time-stamp is denied
    - time-stamps can be used to avoid deadlocks in two-phase locking

Atomic actions / 2.

- Serializability (continued)
  - **resource access graph**: represents executing / committed actions
    - nodes: actions / resources; arcs: action ↔ resource relationships
    - the graph must remain *acyclic* to ensure serializability
    - nodes of terminated actions without incoming edges are deleted
- Recoverability / totality

  Either all of the state modifications take place or none of them.
  The system state reflects only the effects of completely executed actions.

  - **commit operation**: an action makes its state change available

    Irrevocable action, indivisible with respect to failures.

    i. install and restore process state

    ii. commit or abort operation consistently

Atomic actions / 3.

Update strategies:

- In-place updating

  A single copy of each data element is directly modified.

  – **installation of a new state** is trivial

  – **restoring a state**: each data element is restored to its old value

  – realization: maintain an incremental log in stable storage

- Shadow updating

  Two copies of the data element + pointer to access the current element.

  - current copy: most recent committed value
  - shadow copy: updated but not yet committed value

  – **installation of a new state**: current ↔ shadow pointer change

  – **restoring a state** is trivial

Atomic actions / 4.

<u>Commit protocols:</u>

- Two-phase commit protocol
  - one process acts as a *coordinator*, the others are *participants*
    1. send requests, receive local decisions → form final decision
       on commit: write *commit record* to stable storage
    2. multicast final decision, abort / commit, wait for acknowledgments
  - the protocol guarantees, that
    - all processes that reach a final decision reach the same one
    - if all local decisions are commit then the final decision is commit
    - if all failed machines restart then all processes reach a final decision
  - **three-phase commit protocol**: nonblocking property
    - additional buffer phase: removes the WAIT state from the *concurrency set* of the COMMIT state

Resilient processes and stable storage / 1.

A resilient process can continue to execute correctly even if interrupted by a failure and then restarted.

- Recovery techniques
  - **backward recovery**: reconstruct a a consistent and valid past state
    - *checkpoints*: entire state of a process is periodically stored
    - checkpoints must be written atomically with respect to failures
  - checkpoint **timing**: writing a checkpoint $\leftrightarrow$ redoing a computation
    - find a state with a small amount of data to be checkpointed
    - is it possible to restart the computation from every state?
    - can be the values generated by the program as outputs repeated?
  - **stable** $\leftrightarrow$ volatile **storage**: contents remain intact / are lost on a failure
    - access is performed using atomic read and write operations
    - failure resilience: redundant copies on multiple devices, etc.

Resilient processes and stable storage / 2.

Interacting resilient processes:

Process: a sequence of events (local computation, send / receipt of a message).

A process *becomes dependent* on the sender process after receiving a message.

– **system state**: history of events forming the set of all process states
  - *consistent*, if for each message receipt event there is a sending event

– problem: how to maintain a consistent system state despite failures?

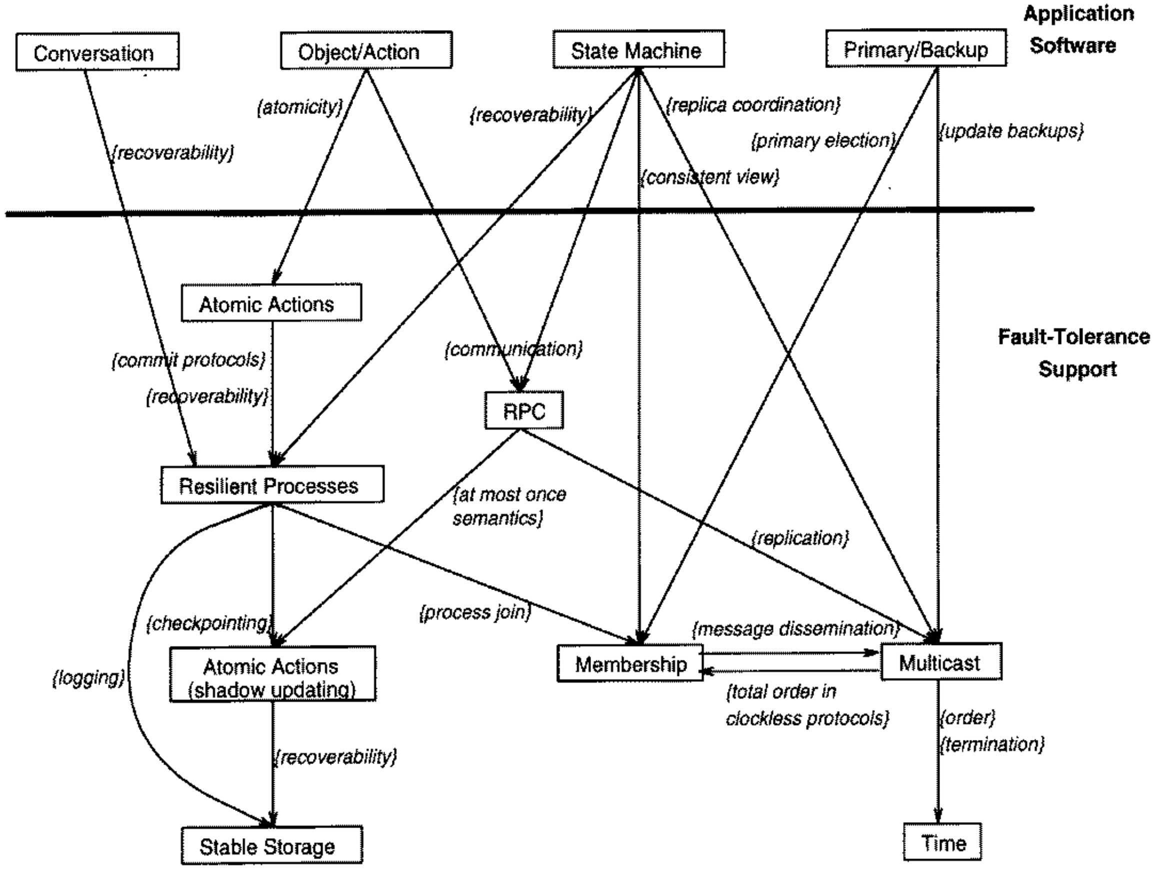
- Checkpointing and rollback recovery

  a. independent periodic checkpointing:
     - processes must dynamically determine a consistent set of checkpoints
     - coordination is required between processes during recovery
     - a rollback may result in a cascade of rollbacks: *domino effect*
     - processes must retain all of their checkpoint indefinitely

Resilient processes and stable storage / 3.

- Checkpointing and rollback recovery (continued)
  b. checkpoint coordination:
    - two-phase commit protocol for generating consistent checkpoints
    - *permanent* and *tentative* checkpoints
    - synchronization is required on rollback
    - two-phase recovery protocol to avoid *livelocks*
- Message logging
  - **optimistic recovery**: processes checkpoint independently and log input messages along with dependency info in stable storage
  - recovery consists of
    i. restoring an earlier state + potentially replay logged messages
    ii. recognize which processes depend on lost states → roll them back
    iii. commit output in a consistent, fault-free system state

Common dependencies / 1.



Common dependencies / 2.

Application level:

- Conversation
  - a collection of *resilient processes* that cooperate by message-passing
- Object / action
  - actions are *atomic*
  - objects interact using *RPC*
- State machine
  - → *multicast*: replica coordination and consistency maintenance
  - → *membership*: to determine the set of functioning replicas
  - → *resilient processes* reintegrate repaired or restarted replicas
  - → *regular* or *replicated RPC*: communication between state machines

Common dependencies / 3.

Application level (continued):

- Primary / backup
  - *multicast*: update the backup processes to reduce recomputation
  - *membership*: recognize the failure of the primary process

Fault-Tolerance Support level:

- Atomic actions
  - *resilient processes*: in-place updating, recovery action upon restart
- Remote procedure call
  - *atomic actions*: orphan removal (at most once execution semantics)
  - *multicast*: reliable, consistently ordered delivery (replicated RPC)

Common dependencies / 4.

- Resilient processes
  - *stable storage*: writing checkpoints and / or log entries
  - *atomic actions*: implement atomicity of writing large checkpoints
  - *membership*: consistent reincorporation of recovered processes
- Atomic actions (shadow updating)
  - *stable storage*:
    - atomic actions at this level are used to manipulate persistent data
    - recoverability requires the data to be stored on ~
- Multicast and membership
  - *common global time*: consistent message ordering, termination
  - *membership*: acknowledgment, funnel process (clockless multicast)
  - *multicast*: reliable dissemination of “I’m alive” messages