Abstractions for Building Fault-Tolerant Distributed Software

Richard D. Schlichting

Department of Computer Science

The University of Arizona

Tucson, AZ 85721, USA

Motivation

Highlight progress over the past 20 years in simplifying construction of *fault-tolerant software* (FTS) for distributed system.

FTS is software that can meet its specifications despite failures in the (virtual) computing platform on which it executes.

Possible lessons for intrusion-tolerant systems (ITS):

- Like FTS, ITS must cope with the unexpected.
- May be possible to use similar solutions, approaches, methodologies, techniques, etc.

Based on the use of *models*, *paradigms*, and *abstractions*:

- Use *failure models* to specify assumptions about effect of failures.
- Organize application around a canonical *software structuring paradigms* for FTS.
- Develop a set of underlying *system abstractions* to support the chosen paradigm.

Just the standard use of abstraction and software layering to simplify software, but especially important for complex software such as FTS.

Failure models for processors:

- Fail-stop: either behaves correctly or stops in a way that is detectable by other processors.
- Crash or fail-silent: either behaves correctly or stops.
- Timing: may produce output outside of specified time interval.
- Byzantine: any behavior is possible; that is, no assumption is made.

Many other failure models have been defined in the literature.

Important: the weaker the assumptions made, the more expensive the solution.

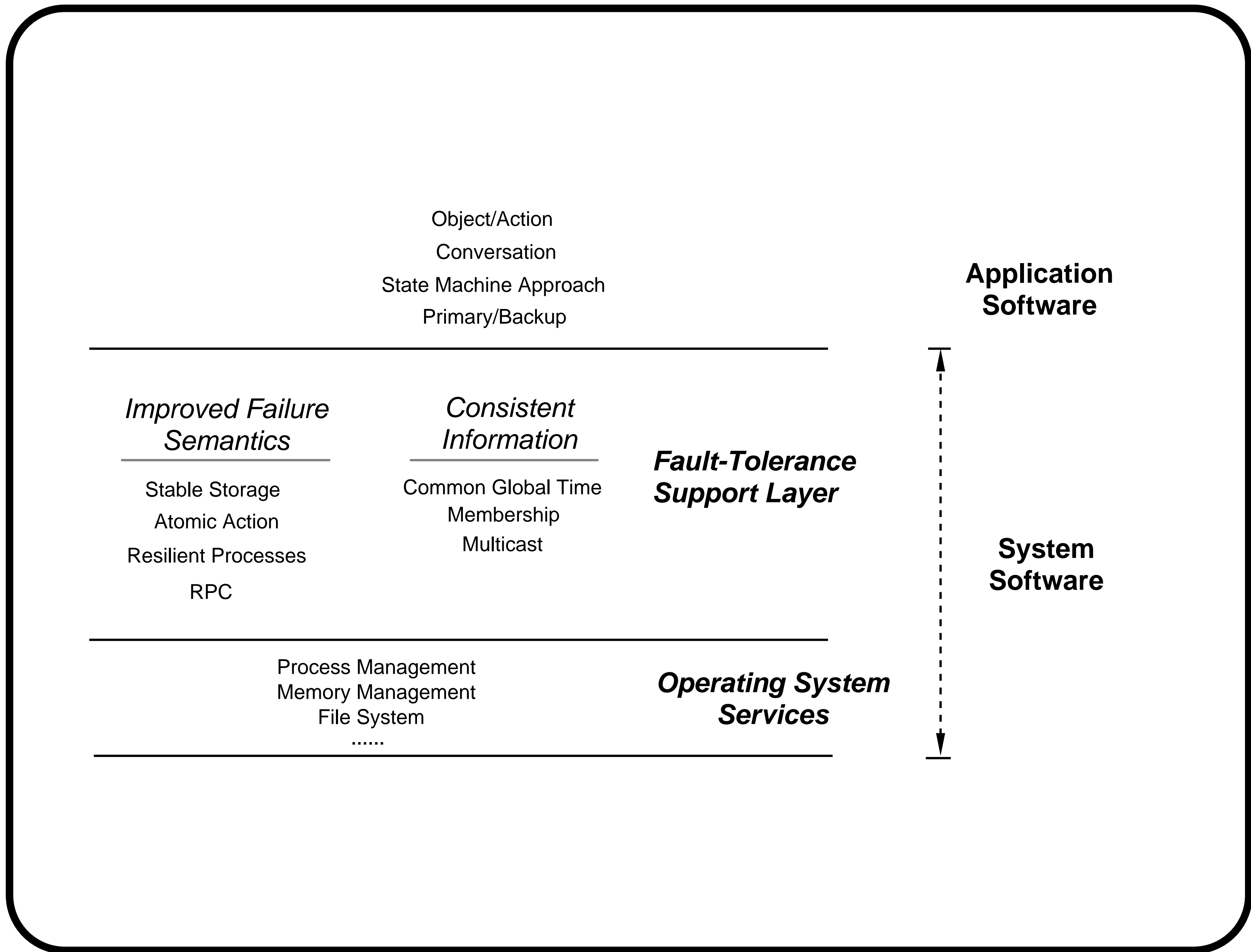
Structuring paradigms:

- Replicated state machine approach: replicated processing.
- Primary/backup: one primary process with backups in case of failure.
- Object/action model: passive objects together with atomic actions.
- Others....

Abstractions:

- Common global time: a single logical time base across all machines.
- Group multicast: send a message to a collection of processes with ordering and atomicity guarantees.
- Remote procedure call: a service request from a client process to one or more server processes.
- Membership: used to maintain consistent information about functioning and non-functioning processes.
- Atomic actions: provides all-or-nothing execution semantics.
- Stable storage: perfect storage that is not affected by failures.
- Resilient processes: a logical process that can continue execution even if interrupted by failure.

Each abstraction either provides improved failure semantics or consistent information in a distributed system.


Object/Action
Conversation
State Machine Approach
Primary/Backup
Application Software
Improved Failure Semantics
Stable Storage
Atomic Action
Resilient Processes
RPC
Consistent Information
Common Global Time
Membership
Multicast
Fault-Tolerance Support Layer
System Software
Process Management
Memory Management
File System
......
Operating System Services

Structuring Paradigms for Fault-Tolerant Software

Common ways to structure fault-tolerant software:

- Replicated state machine approach.
- Primary/backup approach.
- Object/action model.

Choice of which to use depends primarily on the characteristics of the application.

Overview some of the paradigms and their supporting abstractions.

State Machine Approach

An application is structured as a collection of interacting services.

Each service is implemented by a state machine:

- The state machine maintains state variables.
- Upon receiving a *command* in a message, the state machines modifies the variables; it may also generate output.
- Execution of commands must be deterministic and atomic with respect to other commands.
- Implication: the sequence of changes in the state variables and the output are completely determined by the input sequence.

To provide fault tolerance, the state machine is replicated on separate machines in a distributed system.

Each replica processes each command.

Key requirement: Each replica must process input commands in the same order to ensure that the state variables on the different replicas remain consistent.

State machine approach is a formalization of the use of replicated processing for fault tolerance.

Supporting abstractions: common global time, multicast, membership, resilient processes, stable storage.

Primary/Backup Approach

Basic organization is similar to the state machine approach.

Difference is that only one replica called the *primary* executes each command. Other replicas are *backups*.

When the machine executing the primary fails (e.g., crashes), some backup process is designated as the new primary (*failover*).

Starting state of new primary is read from a *checkpoint* generated by the old primary.

Checkpoint can be written to stable storage or actively transmitted to the backups by the primary during execution.

The Object/Action Model

Application structured as a collection of objects:

- Passive entities that encapsulate state.
- State is typically long-lived data on stable storage.
- Export operations that modify state.
- May be located on different physical machines.

Actions are active entities that invoke operations exported by objects to modify the state. May logically move from machine to machine.

Actions are *atomic*, which means that intermediate states of the objects that it modifies are invisible to other actions despite possible concurrent execution and failures.

Atomic execution can be guaranteeed by ensuring:

- Serializability: Effect of executing multiple actions is equivalent to some serial schedule of actions.
- Recoverability: Execution is all or nothing despite failures.

Atomic actions are called transactions in database applications.

Supporting abstractions: stable storage, atomic actions, RPC, resilient processes.

Multicast and Membership

Talk briefly about two useful network services (abstractions): multicast and membership.

Basic functions of these services are straightforward:

- Multicast: send a message to all the processes in a group.
- Membership: maintain consistent information about functioning and non-functioning processes in a group.

However, many details, options, and subtle points to consider.

Multicast Service

Many different multicast services have been designed and implemented, each with different properties.

Common properties the service might guarantee:

- Atomicity: A message is delivered either to all of the correctly functioning members of the group or to none of the members.
- Reliability: A message is delivered to every process in the group, even if it is currently failed.
- Ordering: Messages are delivered to all members of the group in a consistent order.
- Termination: A message is delivered within some known time bound. Requires assuming an upper bound on message delivery time.

All properties must be guaranteed despite failures, based on the assumed failure model

Ordering is a key property for simplifying applications, such as those structured according to the state machine approach.

Different orderings are possible:

- FIFO: Messages from a single source process are delivered in the order they were transmitted.
- Causal: Messages from all source processes are delivered in an order that preserves potential causality (Lamport's "happened-before" relation).
- Total: Messages from all source processes are delivered in the same total order.

Total order is easiest to use, but others can allow more concurrent execution.

Membership Service

Keeps track of changes in group membership, such as failures, recovery, new processes joining, and terminating processes departing.

Delivers messages reporting on changes to the application at each machine.

Membership is often tightly coupled with a multicast service.

- Membership uses multicast to send messages.
- Multicast uses information from membership to order application messages.

Properties that a membership service might guarantee:

- Accuracy: a failure is only reported if the machine has actually failed; similar for recovery.
- Liveness: any failure or recovery is eventually reported.
- Agreement: any change that is delivered to the application on one machine is eventually delivered on all machines.
- Ordering: consistent delivery order on all machines, either with respect to other notification messages or with respect to all messages.
- Partition handling: how the service deals with the situation when groups of machines are functioning, but cannot communicate.

Many variants of last two properties.

Conclusions

Use of failure models, structuring paradigms and associated abstractions simplifies construction of FTS and hence, dependable systems.

Many unresolved issues with using these for FTS, much less ITS.

- Composition and dependency issues.
- System structuring and modularization issues.
- Supporting dynamic adaptation.
- Supporting new types of platforms (e.g., wireless).
- ...

Would really like a common framework for *all* QoS attributes (FT, RT security...) to help deal with combinations and to guide work on conceptual and system issues.