Boeing Space and Intelligence Systems
Communications, Network, and Sensors Systems Engineering

Evaluating Distributed Systems Architectures for Fault-Tolerant Applications

SATURN 2008

james.p.scott@boeing.com
Boeing Space and Intelligence Systems
El Segundo, CA

Abstract

Communications, Networks, and Sensors Systems Engineering

- A large body of experience has been developed within the telecommunications industry with regard to fault-tolerant distributed systems architecture. This presentation focuses on key topics to consider in evaluating a proposed architecture for use in asynchronous, event-driven applications whose system quality attributes include stringent requirements for availability, reliability, and evolvability. A representative list of such topics includes:
  - Thread Scheduling Policy
  - Object Management
  - Message Processing
  - Fault Management and Recovery
  - Graceful Degradation Under Load
  - In-Service Software Upgrades
  - System Forensics
- Architecture and design patterns derived from best practices emerging from the telecommunications industry will be discussed in order to provide additional insight into proven architecture and design of deployed commercial systems. In addition, discussion will be provided as to how these topics and patterns can be applied within the context of the ATAM method of software architecture evaluation.

Leveraging Commercial Best Practices Significantly Reduces Risk

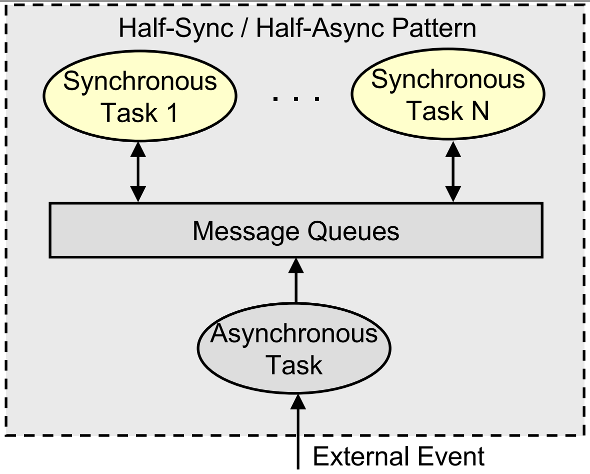
Thread Scheduling Policy

- Typically, these systems are real-time, asynchronous, and event-driven. Correctness and timeliness of response must be predictable.
- Thread Scheduling Models
  - Evaluation Metrics:
    - Scheduler Optimality – an optimal algorithm always achieves a feasible schedule, if one exists
    - Scheduler Stability –ensures that it is possible to predict at design time which tasks will fail in an overload situation
    - Schedulable Utilization – the maximum utilization allowed that achieves a feasible schedule for a given task set
  - Priority-Based Preemption versus Cooperative Scheduling
    - Preemptive scheduler allows scheduler to preempt executing thread in lieu of a pending higher priority thread
      - Static Priority (most commonly used) assigns priorities to threads at design time. Algorithm is stable but non-optimal.
      - Dynamic Priority dynamically updates thread priorities at run-time to reflect changing conditions. Algorithm is optimal but not stable.
      - See Buttazzo, G. Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications. Springer, New York, 2005.
    - Adaptive Partition Scheduling
      - Gracefully degrades performance under load by providing a minimum percentage of CPU throughput to selected groups of threads, even though other pending threads may possess higher scheduling priority. Devolves to priority-preemption for nominal load.
    - Cooperative Scheduling
      - Mitigates risks associated with preemptive scheduling and thread-safety by allowing the application to determine when the task is released
      - See Pont, M. Patterns for Time-Triggered Embedded Systems, Addison-Wesley, Reading, MA, 2001.

Carefully Engineered Limits on Scarce Resources is a Key Design Concept

Thread Pooling Strategies

- Thread Pooling improves performance and stability and negates the overhead associated with thread creation and destruction.
  - Evaluation Metrics:
    - Scalability – Endpoints and event demultiplexing
    - Efficiency – Data movement, context switches, memory allocation, etc
    - Optimality – Stack and thread-specific storage
    - Priority Inversion – Mitigation of priority inversion (bounded and unbounded)



- Use Half-Sync/Half-Async pattern to decouple apps and I/O
  - Decomposes thread pool into three layers: synchronous layer, message queue layer, and async I/O layer
    - See Schmidt and Cranor. "Half-Sync/Half-Async: An Architectural Pattern for Efficient and Well-structured Concurrent I/O" in Pattern Languages of Program Design 2, Addison-Wesley, Reading, MA, 1996.
- Use Leader/Followers pattern to demultiplex I/O events into thread pools without requiring additional I/O threads
    - Provides efficient concurrency model where multiple threads take turns sharing event detection, demultiplexing, dispatching, and processing
    - See Schmidt, et al. "Leader/Followers: A Design Pattern for Efficient Multi-threaded Event Demultiplexing and Dispatching", http://citeseer.ist.psu.edu/731103.html, 2000.

Thread Pooling Improves Response Time and System Determinism

Object Management

- Memory Allocation:
  - Evaluation Metrics:
    - Efficiency – Mitigation of memory fragmentation typical of heap-based allocation
    - Reliability – Mitigation of memory exhaustion due to fragmentation over large mean mission duration
  - Use of *Pool Allocation* Pattern (aka *Object Pooling*)
    - Object Pooling avoids memory fragmentation by allocating blocks in large, contiguous clusters at system init time
    - Object Pool creates large numbers of memory blocks and places them onto free queues. The size of each pool is dimensioned so each application has the memory it requires when the system is operating at peak load.
    - Memory for an object is obtained at run time by dequeueing a block from the object pool associated with a class
    - Design guidelines (See Utas, Greg. Robust Communications Software. Wiley, West Sussex, England, 2005):
      - All subclasses derived from the same framework base class allocate their objects from the same Object Pool
      - For increased efficiency, multiple block sizes may be accommodated by applying a *Best-Fit* allocation algorithm
      - A small set of configurable parameters (max number of user sessions, network diameter, etc) are used to dimension the Object Pools
      - Allocating Object Pool blocks in an array of linked lists (Class x Size) allows Object Pools to be redimensioned dynamically
  - Use of *Smart Pointer* Pattern
    - Common pointer processing pathologies (memory leaks, uninitialized ptrs, dangling ptrs, and defects in ptr arithmetic).
    - Smart Ptrs are object, allowing proper construction, initialization, access, and destruction to be verified at run time.

Design Limits for Memory are Dimensioned per Peak Loading Scenario

Message Processing

- Reliable message delivery is provided for all inter-process communication (IPC):
  - Design options:
    - Reliable User Datagram Protocol (RUDP)
      - Thin Client-Server based transport with message acknowledgment, window-based congestion control, and server-side retransmission (see RFC 1151 for additional information)
    - Streams Control Transmission Protocol (SCTP)
      - Supports error-free, sequenced transmission over redundant links with optional message bundling (see RFC 2960 for additional info)
      - Stewart and Xie. Stream Control Transmission Protocol: A Reference Guide. Addison-Wesley, Boston, MA, 2002.
    - Use of Additive Increase Multiplicative Decrease (AIMD) protocols such as TCP are ill-suited for internal transport
- Message Formatting and Processing:
  - Use of Type/Length/Value (TLV) format simplifies message parsing and increases capacity in both space and time dimensions
    - Use of a Fence Pattern allows system to detect applications that overwrite the memory area allocated for a parameter
      - Fence Pattern (0xDEADBEEF, for example) is placed immediately at the end of the memory allocated for a given parameter
  - Parameter Typing
    - Base class builds and parses TLV messages by defining functions that add, find, and iterate over parameters, allowing derived classes to build and parse messages using a strong typing, leading to increased reliability.
      - See Utas, Greg. Robust Communications Software. Wiley, West Sussex, England, 2005.

Emphasis for Message Processing is on Reliability and Reduced Complexity

Management of Software Faults

- Fault-Tolerant distributed systems typically employ hierarchical fault management to provide durable operation across system faults. A top-level Fault Manager is responsible for correlating detected faults, managing the recovery process, and notifying the operator.
  - Evaluation Metrics:
    - Reliability – the probability that the system is *operational* at a given point in time (the expected value of the failure PDF)
    - Steady State Availability – the limit as $t \to \infty$ of the probability that the system is *working correctly* at time t
  - Software Fault Management Mechanisms
    - Use of defensive coding practices (run-time validation of arguments, pointers, message format, return values, etc)
    - Selection of a microkernel-based operating system that supports task-level memory protection and restart
    - Thread class is defined to allow for exceptions and signals to be processed without crashing the system
    - System monitor audits system for resource pool and mem usage, runaway tasks, inconsistent state, excessive events
    - Use of an Escalating Restart policy (listed in order of increasing service affectation)
      - Warm Restart / Level 0 – all child threads are killed and recreated (only data associated with child threads is freed and reinitialized)
      - Warm Restart / Level 1 – all unprotected memory is freed and reinitialized (protected memory remains untouched)
      - Warm Restart / Level 2 – all memory (protected and unprotected) is freed and reinitialized, forcing all applications to reload and reinitialize
        - Note: Performance of an L2 Warm Restart can be improved by periodically saving a bin image of the database atomically to NVRAM
      - Cold Restart (aka Reboot) – software is reloaded and system is reinitialized

Defensive Coding and Run-Time Monitoring Improve System Reliability

Management of System Faults

- Mitigation of hardware or system faults (beyond software defects) employs redundant sparing and failover models (aka *Equipment Protection*):
  - Cold Sparing
    - Inactive standby processor assumes role of active role upon failure of active processor
      - Failure transparency is limited since all work in the failed active processor is lost during the recovery
      - Recovery time is slow since the standby processor must be powered-on and configured prior to assuming the active processor role
      - Cold sparing primarily improves reliability (due to cold sparing), rather than availability (due to lack of outage time and service affectation)
  - Warm Sparing
    - Active standby processor maintains loosely coupled state with active processor, assumes role of active upon failure
      - Active processor sends periodic checkpoint data to warm spare in order to achieve loosely coupled synchronization of system state
      - Improves failure transparency (compared to cold sparing) to the extent that checkpointing techniques maintain synchronization
      - See Elnozahy, Alvisi, Wang, and Johnson. “A Survey of Rollback-Recovery Protocols in Message-Passing Systems” for additional.
  - Hot Sparing
    - Active standby processor maintains tightly coupled state with active processor, assumes role of active upon failure
      - Both processors process identical messages, but only the active processor interacts with peering systems/subsystems
      - Provides complete transparency to system failures
      - Typical commercial solutions employ lock-stepped processors and mirrored memory, which increases design cost and complexity (compared to warm sparing)
      - See Jalote, P. Fault Tolerance in Distributed Systems. Prentice-Hall, Englewood Cliffs, NJ, 1994 for additional.

Redundant Sparing Techniques Address System Availability Requirements

Graceful Degradation Under Load

- Fault-Tolerant distributed systems must gracefully degrade performance under heavy load while continuing to support high priority sessions and service requests
  - Evaluation metrics:
    - Processor throughput as a function of offered load
    - Multi-service message delay under load
    - Multi-service message loss under load
  - Design Goals:
    - Transient overload is non-service affecting. Services degrade gracefully under persistent overload.
  - Design Techniques:
    - Use of classification and multi-service queues for arriving/pending work
      - High-Priority Work (Encompasses both arriving/pending, and in-progress work)
      - In-Progress Work (Completion of work that has already been started)
      - Arriving/Pending Work (Newly arriving work requests that are not classified as high-priority)
      - Suggested service discipline is Modified Deficit Round Robin (MDRR). See Shreedhar and Varghese. “Efficient Fair Queueing Using Deficit Round Robin”. IEEE Transactions on Networking, June, 1996.
      - Discard policy can employ either drop-tail or drop-front, depending on expected application and user behaviors
    - Use token bucket filters to rate limit arriving workload
      - Service processor determines rate at which tokens are added to the token bucket filter (rate limiting and backpressure mechanism)

Robust Designs Gracefully Degrade Performance For Persistent Overload

In-Service Software Update Strategy

- Design goal is to achieve non-service affecting in-service software upgrade capability
  - Evaluation Metrics / Requirements:
    - Mean and maximum service outage
    - Version compatibility (backward and forward) which includes database schema migration
    - Network security and software image integrity (support for authentication, encryption, and non-repudiation)
- Types of software updates:
  - Patch – Used to deliver bug fixes. Also referred to as a *dot release*.
    - Function Patching
      - Employs an incremental loader to store updated function into preallocated segment of memory. New version of function employs entry and exit points of old version of function. Also requires that symbol table be updated and cache to be invalidated upon loading.
    - Class Patching
      - Definition of a backdoor mechanism that allows a class to add member data and functions (see previous Utas reference for additional).
  - Upgrade – Used to deliver new features and capabilities. Also referred to as a *dot zero release*.
    - Hitless Upgrade
      - Employs failover to warm or hot spare to execute software upgrade and associated schema changes while minimizing service disruption.
    - Rolling Upgrade
      - Updates constituent components of distributed serially, rather than in parallel. Typical order of rolling upgrade is: 1) administrative node, 2) control node, 3) all service interface host nodes, and 4) all service host nodes.

Hitless In-Service Software Upgrade is Essential for High-Availability

System Forensics

Communications, Networks, and Sensors Systems Engineering

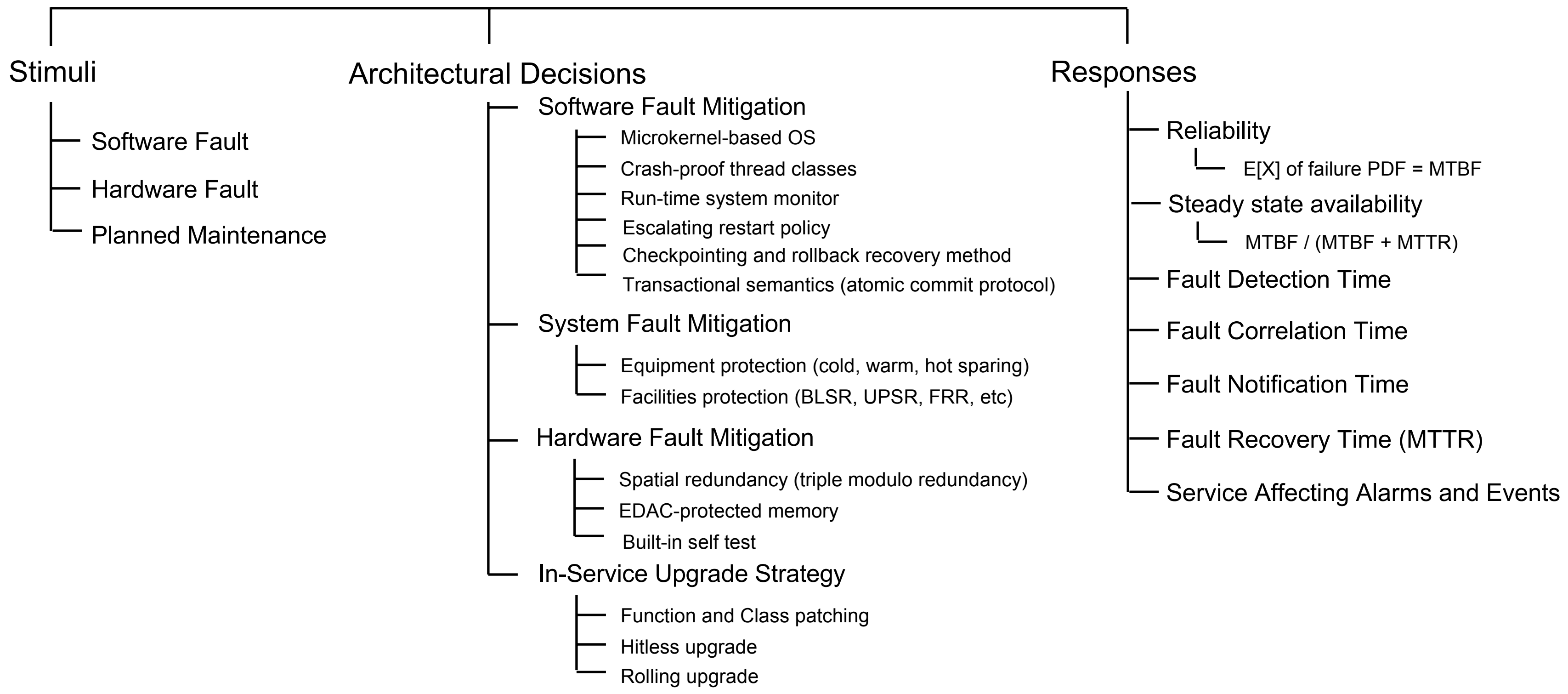
- Run-time debug facilities are essential to maintaining deployed distributed systems
  - Log facilities
    - Software error log – provides information regarding traps (signals and exceptions)
      - System name, processor ID, timestamp, executing thread, exception detected, stack trace of executing thread
    - Software warning log – provides information regarding non-fatal errors
      - System name, processor ID, timestamp, executing thread, function or method call chain
    - Object dump – logs the content of objects cleaned up as the result of a signal or exception (similar to a core dump)
      - Top-level thread walks the list of objects involved in the faulty transaction and invokes their Display methods to create a log
    - Flight recorder – ensures that system logs persist across system restarts
      - Include any error or warning log that set an alarm, informational logs that cleared an alarm, and any service-affecting operations
  - Trace tools (disabled by default)
    - Function trace – captures a record of each function (or method) called
      - Trace banner includes system name, processor ID, timestamp, executed threads, and time spent in each thread
    - Message trace – captures messages transmitted and received by the system
      - Interleaves its records with Function Trace tool when both trace tools are enabled concurrently
    - Tracepoint debugger – uses tracepoints rather than breakpoints for debugging
      - Uses predefined trace instructions to capture run-time system state to a memory buffer.
  - See Utas, Greg. Robust Communications Software. Wiley, West Sussex, England, 2005.

Field-Safe System Forensic Tools are Essential for Deployment

Applicability to the ATAM Method

Communications, Networks, and Sensors Systems Engineering

- We have shown some essential design patterns and associated evaluation metrics used in the design of commercially deployed telecommunications systems. The following is an example characterization of the *Availability* quality attribute:


Stimuli
Software Fault
Hardware Fault
Planned Maintenance
Architectural Decisions
Software Fault Mitigation
Microkernel-based OS
Crash-proof thread classes
Run-time system monitor
Escalating restart policy
Checkpointing and rollback recovery method
Transactional semantics (atomic commit protocol)
System Fault Mitigation
Equipment protection (cold, warm, hot sparing)
Facilities protection (BLSR, UPSR, FRR, etc)
Hardware Fault Mitigation
Spatial redundancy (triple modulo redundancy)
EDAC-protected memory
Built-in self test
In-Service Upgrade Strategy
Function and Class patching
Hitless upgrade
Rolling upgrade
Responses
Reliability
E[X] of failure PDF = MTBF
Steady state availability
MTBF / (MTBF + MTTR)
Fault Detection Time
Fault Correlation Time
Fault Notification Time
Fault Recovery Time (MTTR)
Service Affecting Alarms and Events

The Commercial Design Approaches Discussed Feed Directly Into An ATAM

References

- The following references provide information and design patterns essential to realizing a fault-tolerant distributed system:

  - Buttazzo, G. Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications. Springer, New York, 2005.
  - Cheng, A. Real-Time Systems: Scheduling, Analysis, and Verification. Wiley, Hoboken, NJ, 2002.
  - Douglass, Bruce Powel. Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns. Addison-Wesley, Boston, MA, 1999.
  - Douglass, Bruce Powel. Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems. Addison-Wesley, Boston, MA, 2003.
  - Elnozahy, Alvisi, Wang, and Johnson. “A Survey of Rollback-Recovery Protocols in Message-Passing Systems”. Carnegie Mellon University, CMU-CS-99-148, 1999.
  - Jalote, P. Fault Tolerance in Distributed Systems. Prentice-Hall, Englewood Cliffs, NJ, 1994
  - Hanmer, Robert. Patterns for Fault Tolerant Software. Wiley, West Sussex, England, 2007.
  - Laplante, Phillipe. Real-Time Systems Design and Analysis. IEEE Press, New York, 2004.
  - Pont, M. Patterns for Time-Triggered Embedded Systems, Addison-Wesley, Reading, MA, 2001.
  - Utas, Greg. Robust Communications Software: Extreme Availability, Reliability, and Scalability for Carrier-Grade Systems. Wiley, West Sussex, England, 2005.

- The ATAM method for evaluating software architecture is described in:

  - Clements, Kazman, and Klein. Evaluating Software Architectures: Methods and Case Studies. Addison-Wesley, Boston, MA, 2002.

Author Bio

Communications, Networks, and Sensors Systems Engineering

- James Scott is a Boeing Associate Technical Fellow and has over 20 years of experience in the development of fault-tolerant distributed systems architectures. James currently serves as the chief engineer for the TSAT Digital Processor Subsystem, which is the collection of computers, cross-connects, cryptographic processors, MODEMs, optics, routers, and switches used to form the network nodes for the space-based backbone of the Global Information Grid (referred to as the Transformational Satellite Communications System, or TSAT). Prior to joining Boeing in 2003, James served in chief architect/chief scientist roles for both venture-backed networking startups such as Calient Networks, Geyser Networks, and Polaris Networks, and for publicly traded companies such as Network Equipment Technologies and Nortel Networks. James has degrees from Baylor University and The University of Texas at San Antonio and post-graduate certificates from CalTech and UCLA.

BOEING

Boeing Space and Intelligence Systems

Communications, Networks, and Sensors Systems Engineering
El Segundo, CA USA