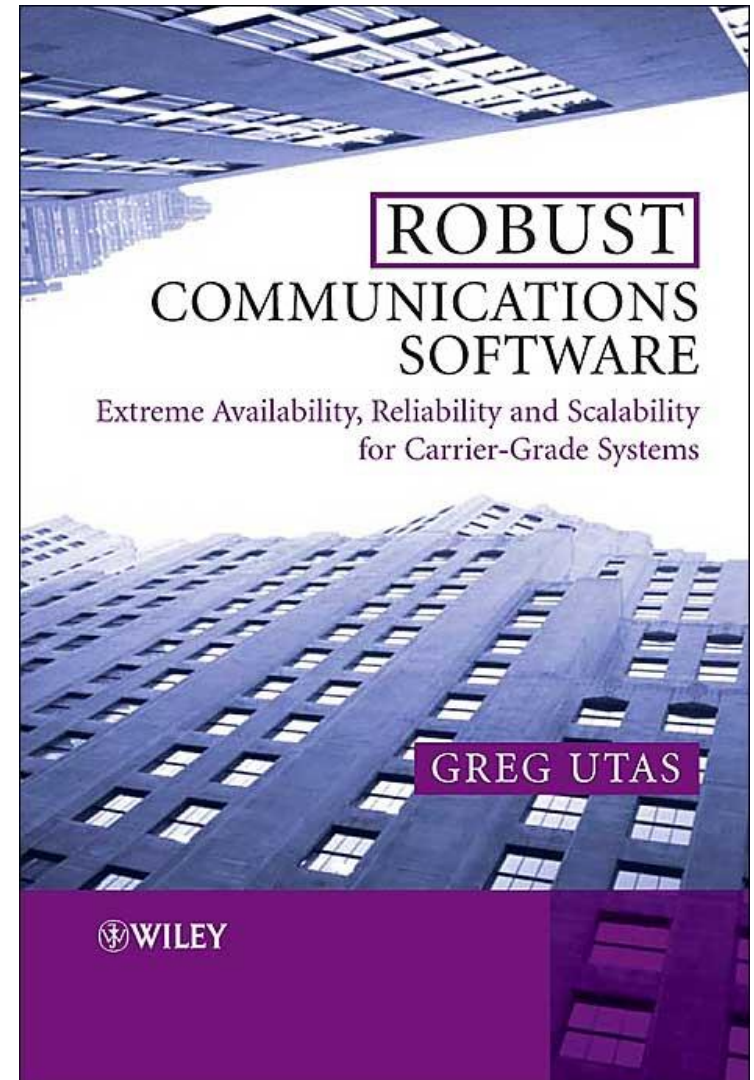# Robust Communications Software

Greg Utas
Pentennea
greg@pentennea.com

Personal Background

❖ software architecture

❖ Nortel (20 years)

  ➢ call server frameworks

  ➢ GSM MSC rearchitecture

❖ Sonim (2 years)

  ➢ push-to-talk for wireless networks

❖ Pentennea

  ➢ carrier-grade software consulting

ROBUST COMMUNICATIONS SOFTWARE

Extreme Availability, Reliability and Scalability for Carrier-Grade Systems

GREG UTAS

WILEY

Introduction

❖ Definitions

❖ Becoming Carrier-Grade

❖ Programming Models

❖ Extreme Software Techniques

Definitions

❖ carrier-grade = meets the strict quality requirements expected by carriers

❖ carrier = operator of a communications network

❖ extreme system = carrier-grade system

❖ extreme software = carrier-grade software

  ➢ *extreme* refers to requirements

  ➢ has nothing to do with "extreme programming"

Becoming Carrier-Grade

❖ focusing on software is necessary but not sufficient

- ➢ also process, hardware, modeling, engineering, testing, documentation, ease of operation

❖ takes time

- ➢ evolve system based on thorough testing and problems uncovered in the field
- ➢ more of a journey than a destination

Programming Model

❖ a set of techniques that a system's software components follow to yield consistent design and behavior

❖ architecture focuses on *interfaces*

❖ programming model focuses on *implementation*

❖ an extreme system needs a programming model

Programming Model Example

❖ a common programming model might include

➢ heap-based memory allocation

➢ garbage collection

➢ synchronous remote procedure calls

➢ preemptive scheduling with semaphores

➢ priority scheduling

➢ spawning numerous short-lived threads

➢ virtual memory

❖ this programming model happens to be dead wrong for extreme systems

Extreme Software Techniques

❖ requirements and characteristics of extreme systems lead to software techniques that

➢ are not common practice in the computing industry

➢ involve trade-offs (e.g. capacity vs. reliability)

➢ do not necessarily apply if one or more of the requirements or characteristics are absent

Extreme System Requirements

❖ Availability

❖ Reliability

❖ Scalability

❖ Capacity

❖ Productivity

Availability

❖ percentage of time that system is in service

❖ ideal is 24 x 7 x 365

❖ acceptable is 99.999% ("five nines")

➢ less than 5 minutes downtime per year

➢ no outage over 30 seconds—planned or unplanned

❖ cannot be shut down for "routine maintenance"

❖ five-nines software, not just hardware

❖ requires duplication of critical components

Reliability

❖ correct behavior while system is in service

❖ ideal is bug-free operation

❖ acceptable is 99.99%

➢ software fault drops 1 session in 10,000

➢ if a session has 10 transactions, this is five-nines with respect to transactions

❖ requires comprehensive testing

❖ robustness = availability + reliability

➢ despite intermittent hardware and software faults, or even errors by system administrators

Scalability

❖ adding more processors supports more users

❖ ideal is linear growth

❖ cannot hit brick wall (asymptotic growth)

❖ 300K users or more per system (shelf)

❖ MSC (mobile switch)

> ➢ assume 900K calls/hour and 24 events/call
>
> ➢ that's 21.6M events/hour = 6K events/sec

❖ requires moving some users or functions to other processors

Capacity

❖ throughput of a single processor

❖ system needs to be cost-effective in terms of processors and memory

❖ new software release can't degrade capacity more than 3%

❖ requires capacity measurements, modeling, and recovery

Productivity

❖ effectiveness of development team

❖ an implicit, rather than an explicit, requirement

➢ millions of lines of software

➢ hundreds of engineers

❖ requires focusing on software excellence

➢ architecture and programming model

➢ application frameworks

➢ tools for debugging and testing

❖ productivity extends product's lifetime

➢ more time for return on investment

Extreme System Characteristics

❖ Embedded

❖ Reactive

❖ Stateful

❖ Real-Time

❖ Distributed

❖ Reference Model

Embedded

❖ dedicated to a specific purpose

❖ can therefore be modelled, studied, and tuned

❖ often contains custom hardware or software

Reactive

❖ responds to external inputs from users

❖ protocols are central to reactive systems

❖ a protocol is defined by

  ➤ a set of signals (message types) and parameters

  ➤ for each signal, rules about the parameters that are mandatory, optional, or illegal

  ➤ rules about the order in which signals may be sent and received

❖ a message contains a signal and parameters

Stateful

❖ must maintain state information to handle inputs correctly

❖ state machines are central to stateful systems

➤ a set of states

➤ a set of events (inputs)

➤ a set of event handlers that perform the work associated with legal state-event combinations by

- sending messages (outputs)
- updating the state
- raising another event

Real-Time

❖ need to perform work before a deadline

➢ hard real-time: being late causes failure

➢ soft real-time: being late causes dissatisfied users

❖ hard real-time software

➢ close to the hardware

➢ modest code bulk (e.g. device drivers)

❖ soft real-time software

➢ close to the user

➢ significant code bulk (e.g. applications)

❖ much of this tutorial is geared at soft real-time

➢ techniques for hard and soft real-time may differ

Distributed

❖ more than one processor

❖ processors communicate with messages

❖ serves various purposes

  ➢ meeting scalability requirements

  ➢ meeting availability requirements (replication)

  ➢ simplifying design (separating hard real-time software from soft real-time software)

❖ simplifies local complexity by focusing work

❖ increases global complexity

  ➢ deciding how to partition work is only the first problem

Reference Model

administration node

service nodes

control node

access nodes
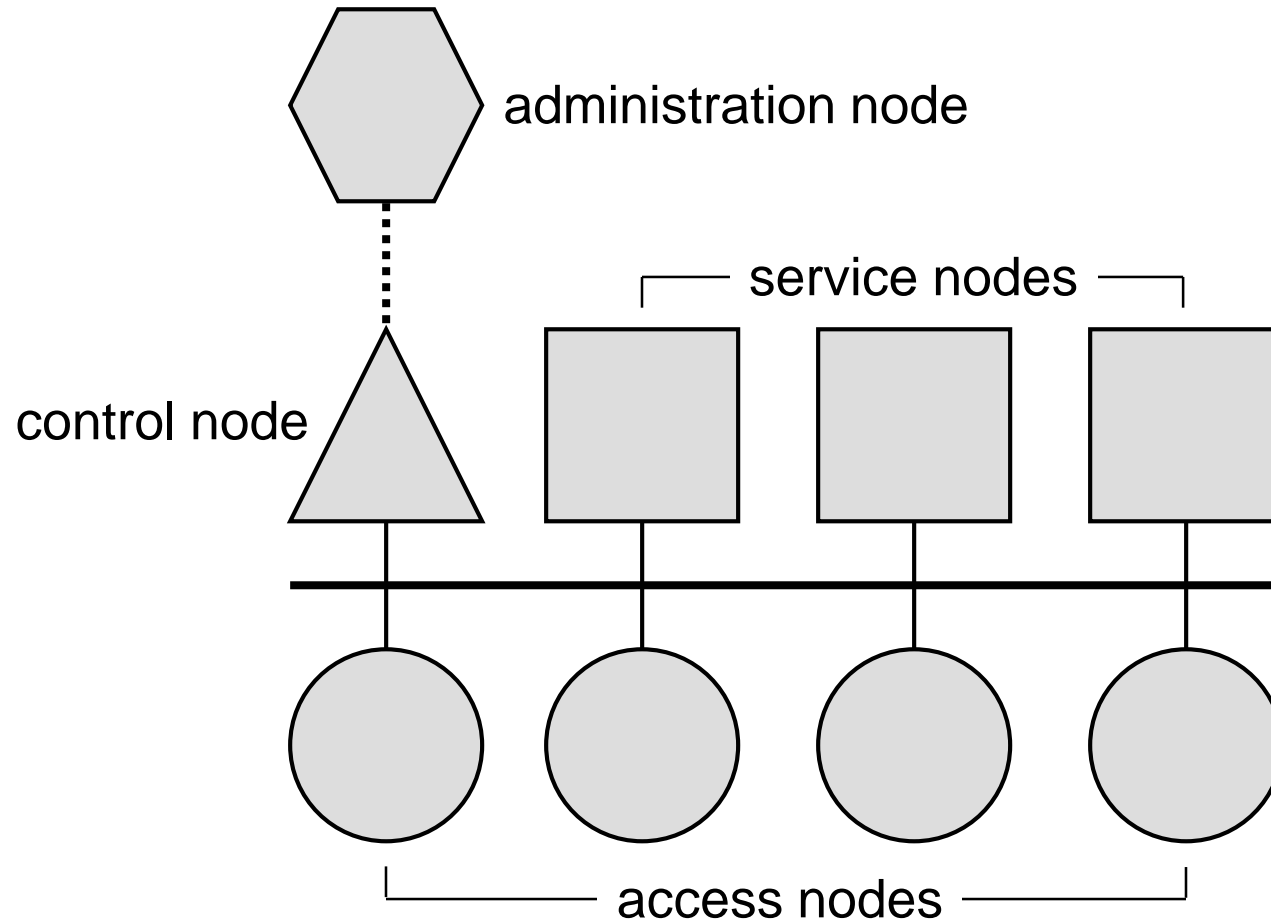
Object Orientation

- ❖ Rationale for Object Orientation
- ❖ Choosing a Language
- ❖ Limiting Distractions
- ❖ Achieving Efficiency
- ❖ Improving Efficiency
- ❖ Object Management
- ❖ Object Nullification

Rationale for Object Orientation

❖ fundamentals

➢ encapsulation: hiding data

▪ limits amount of rework when data structures change

➢ polymorphism: same interface, different implementations

▪ vital for defining frameworks

➢ inheritance: same implementation with additions

▪ significantly reduces amount of code cloning

❖ applications and their frameworks need object orientation

➢ >95% of a large extreme system

❖ use C only for hard real-time software

Choosing a Language

❖ criteria

➤ type-checked

➤ known to many software engineers

➤ available on many platforms

➤ efficient enough to meet capacity requirements

➤ allows control of low-level functions

❖ by these criteria, C++ is the obvious choice

❖ could choose Java if the team insists *and*

➤ capacity will be acceptable (probably compiled)

➤ you don't need control of low-level functions (or you can use the Java Native Interface to provide them)

Limiting Distractions

Distraction	Comment
multiple inheritance	use (if at all) only to mix in pure interfaces; use aggregation to combine concrete classes
operator overloading	an operator is just a function invoked using a different syntax
templates	use judiciously: bloat and customization issues
exceptions	use sparingly, for "fatal" errors only
methodologies	be eclectic; avoid religious debates
notations	of marginal value except to describe frameworks
training	a few designers can shepherd a large team
capacity impact	marginal—*if given adequate attention*

Achieving Efficiency

❖ most overhead is from creating and destroying objects (calls to constructors and destructors)

  ➢ avoid

   ▪ declaring objects on the stack

   ▪ passing objects by value

   ▪ copying objects

  ➢ reduce number of objects by merging classes that are not orthogonal

   ▪ improve cohesion—but preserve loose coupling

  ➢ use singletons and flyweights where possible

❖ in-line "get" functions for popular data members

❖ beyond this, overhead should not be an issue

Improving Efficiency

❖ various techniques can streamline object construction in specific situations

➢ Embedded Object (placement `new` in `int` array)

➢ Object Template (block copy to speed construction)

➢ Quasi-Singleton (*usually* acts like a singleton)

➢ Object Morphing (change subclass at run-time)

❖ these techniques add some complexity

➢ only use them to meet a non-negotiable benchmark

➢ morphing is sometimes elegant for other purposes, however

Object Management

	Heap ✖	Object Pools ✔
Synopsis	objects allocated from a common heap	objects allocated from pools created during system initialization; each pool is associated with an abstract base class
Capacity impact	moderate	low
Memory impact	low—but fragmentation is a serious risk	moderate (pool must satisfy largest subclass)
Gobbler impact	could exhaust heap, which crashes system	pool sizes engineered; allocation eventually stops
Garbage collection	none (risk of leaks) or a foreground task (difficult and expensive)	background (a corrective audit when applications forget to delete objects)

Object Nullification

❖ reset object's memory to fixed pattern (e.g. `0xdfdfdfdf`) upon deletion

❖ purpose is to detect accesses to deleted objects

❖ in lab, default should be to reset entire object

❖ in field, default should be to reset `vptr` only

Basic Design Patterns

- ❖ **`Object`** Class
- ❖ Singleton
- ❖ Registry
- ❖ Flyweight
- ❖ Polymorphic Factory

`Object` Class

❖ define an `Object` class that all non-trivial classes ultimately derive from

❖ contains basic functions, such as

➤ **`ClassName`**: print name of object's class

➤ **`Display`**: print all data members

➤ **`GetSubtended`**: add owned objects to a list

Singleton

❖ the only instance of its class

❖ used when

➢ more than one instance would cause incorrect behavior

▪ **Heap** (manages global free memory)

➢ creating copies would be inefficient

▪ each **State** (maps an **Event** to an **EventHandler**)

❖ created during system initialization

➢ avoids overhead after system starts to process work

Registry

❖ selects an object using an identifier

➢ identifier is often an array index, in which case the registry is an array

➢ identifier may correspond to the object's class

❖ Registry often contains Singletons or Flyweights

Flyweight

- ❖ shared by all would-be objects that have the same member data
  - ➤ are therefore read-only (stateless)
- ❖ used when creating copies would be inefficient
- ❖ usually placed in a Registry
  - ➤ **Font** (Registry) of **Character** (Flyweights)
  - ➤ **Font** attributes include
    - ▪ **height_** (point size)
    - ▪ **chars_** (array of **Character**, indexed by **CharacterId**)
  - ➤ **Character** attributes include
    - ▪ **width_** (in a proportional font)
    - ▪ **image_** (bitmap rendering of the character)

Polymorphic Factory

❖ creates an object whose subclass is not visible to the user of the factory

❖ used to preserve layering and partitioning

➢ motivated by "**switch** Considered Harmful"

❖ example of creating a Card subclass:

```
CardId cid = someCardId;
auto reg = Singleton<CardRegistry>::Instance();
Card *c = reg->CreateCard(cid);
...return factories_[cid]->Create(); // CreateCard
   ...return new SomeCard();        // Create
```

➢ the above replaces **switch(cid)...** in a file that **#includes** the world

Threads

❖ Context Switching

❖ Key Messaging Alternatives

❖ Half-Sync/Half-Async

❖ Thread Selection

❖ Blocking Operations

Context Switching

	Preemptive Scheduling ✖	Cooperative Scheduling (Run to Completion) ✔
Synopsis	scheduler preempts threads at any time	thread decides when it will yield: `MutexOn`, `MsecsLeft`, `Pause`, `MutexOff`
Semaphores	ubiquitous, to protect critical regions	one long semaphore (after each transaction or every n repetitions of a high level loop)
Capacity impact	moderate: time wasted on context switching	low (few threads; one thread can handle all transactions; switching only occurs when necessary)
Issues	*highly* error prone: "p() and v() Considered Harmful"	sanity timeout required to guard against infinite loops; precludes synchronous messaging (RPCs)

Key Messaging Alternatives

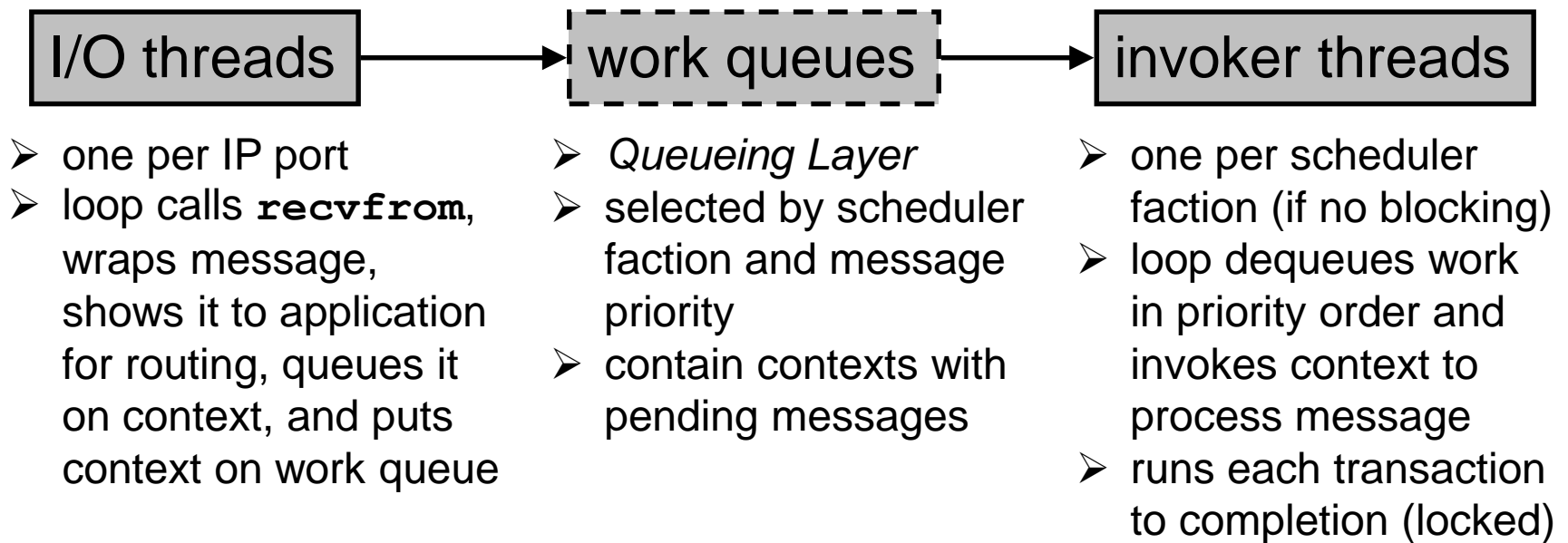|	Synchronous Messaging ✖	Asynchronous Messaging ✔
Synopsis	send message and wait for reply inline; other inputs queued until reply or timeout	send message and save state; inputs other than reply can still arrive
Capacity impact	high: thread pool with context switching on sends	low: only one thread required to run state machines
Issues	deadlock; latency caused by semaphores, timeouts and serial processing ("hourglass" syndrome); state machines do not exist; replies routed differently than other inputs; RPCs explicitly name destination methods, which results in poor decoupling	developing state machines takes more time because of additional state-event pairs; race conditions (need priority messaging to support run-to-completion within a cluster of state machines in the *same* processor, and maybe the ability to defer inputs in the *interprocessor* case)

Thread Selection

	Priority Scheduling ✕	Proportional Scheduling ✓
Synopsis	each thread has a priority	each thread has a faction
Context switching	thread preempted when a higher priority thread is ready to run	scheduler driven by a timewheel that specifies the faction whose threads can run during a given timeslice
Issues	thread starvation; priority inversion; playing games with priorities or number of thread instances to get more processor time	engineering the timewheel to keep the system balanced during peak load; customizing third-party scheduler

❖ Proportional scheduling is for soft real-time work. Factions include
  ➢ *Priority* for hard real-time work: should be minimized, but always runs first
  ➢ *Payload* for session processing: gets most of the CPU time (75%+)
  ➢ *Maintenance* for handling system faults: gets more time during catastrophe
  ➢ *I/O* for receiving messages: ceiling imposed by disabling I/O interrupts

Half-Sync/Half-Async

❖ separate I/O from applications to

  ➢ prioritize incoming work

  ➢ eliminate I/O blocking in applications

  ➢ separate protocols from applications

I/O threads	→	work queues	→	invoker threads

I/O threads
- ➢ one per IP port
- ➢ loop calls `recvfrom`, wraps message, shows it to application for routing, queues it on context, and puts context on work queue

work queues
- ➢ *Queueing Layer*
- ➢ selected by scheduler faction and message priority
- ➢ contain contexts with pending messages

invoker threads
- ➢ one per scheduler faction (if no blocking)
- ➢ loop dequeues work in priority order and invokes context to process message
- ➢ runs each transaction to completion (locked)

Blocking Operations

❖ locked thread must release lock during a blocking operation

  ➢ provide functions for this purpose

❖ the general structure of a locked thread is

```
MutexOn();
    while(true)
    {
        perform some work;
        EnterBlockingOperation();
            perform blocking operation;
        ExitBlockingOperation();
        perform some more work;
        if(MsecsLeft() < threshold)
            Pause(0); // let other work run
    }
MutexOff();
```

Distribution

❖ Benefits of Distribution

❖ Drawbacks of Distribution

❖ Key Distribution Alternatives

❖ Hierarchical Distribution

❖ Symmetric Multi-Processing

❖ Half-Object Plus Protocol

Benefits of Distribution

❖ achieving scalability

  ➢ add processors to handle more work

❖ improving reliability

  ➢ extra processors take on work of failed processors

❖ maintaining simplicity

  ➢ separate hard real-time software from soft real-time software

❖ accessing centralized resources

  ➢ when distributing them is too difficult or costly

  ➢ large databases, specialized hardware

Drawbacks of Distribution

❖ distribution increases complexity

➢ partial failures (another node dies)

➢ transient states (while waiting for responses)

➢ increased latency (another node is busy)

➢ timeouts (another node is too busy)

➢ reduced capacity (building, sending, receiving, and parsing messages)

➢ protocol backward compatibility (different releases)

❖ consequently

➢ distribution is not an end in itself

➢ there is *no such thing* as the "transparent distribution of objects"

Key Distribution Alternatives

	Homogeneous ✓	Heterogeneous ✗
Synopsis	move some *users* to other nodes: each node does everything for a subset of users	move some *functions* to other nodes: each node performs a specialized function (somewhat *antithetical* to distribution)
Capacity impact	low (this is how networks as a whole must be designed for scalability)	low (if offloaded work takes a lot of time) to high (if it isn't much more than the messaging overhead)
Issues	none beyond those inherent in distribution	poor decoupling if messages need to pass a lot of information; managing different software loads; increased latency (beyond what already results from homogeneous distribution)

Hierarchical Distribution

❖ the combination of homogeneous and heterogeneous distribution

❖ horizontally homogeneous

  ➢ for reasons of scalability or reliability

❖ vertically heterogeneous

  ➢ for reasons of simplification or centralization

❖ common for a system (shelf) as a whole

Symmetric Multi-Processing

❖ a group of processors share a common memory

❖ generally, requires ubiquitous semaphores (same problem as with preemptive scheduling)

❖ to avoid the problem

  ➢ assign each processor a large, private segment

  ➢ only use shared segments for

   ▪ interprocessor messaging

   ▪ centralized databases

Half-Object Plus Protocol

❖ fully or partially replicating the master instance of an object (e.g. routing table) in other processors

❖ alleviates many drawbacks of distribution

➢ reduces messaging

  ▪ unless data changes more often than users access it

➢ users of object are no longer burdened with timeouts and transient states

➢ reduces latency

❖ also has drawbacks of its own, however

➢ increases memory usage

➢ user obtains a "wrong answer" before a change in the master gets propagated to the local copy

Protecting Against Faults

❖ Defensive Coding

❖ Low-Order Page Protection

❖ Stack Overflow Protection

❖ Data Protection

Defensive Coding

❖ check for **nullptr** and out-of-bound array indices

❖ more generally, defensive coding includes

➢ checking arguments

➢ checking updated or returned values

➢ checking incoming messages

➢ running timers while waiting for responses

❖ needs to be part of development culture

➢ 90% of code in extreme systems is for error paths

Low-Order Page Protection

❖ protect pages 0-*n* to prevent trampling if `nullptr` plus offset is used for write operation

❖ usually protect against read operation as well, to prevent reading of garbage

Stack Overflow Protection

❖ generate signal or exception if thread stack overflows

❖ implemented by putting a protected page beyond the stack

Data Protection

	User Spaces ✗	Write-Protected Memory ✓
Synopsis	assign a process its own memory segment that cannot be accessed by other processes	write-protect critical data that changes infrequently (system configuration, user profiles, many Singletons)
Capacity impact	moderate (remap memory on context switches) to extreme (if data must frequently be accessed using messages)	low (must unprotect memory to change data—but direct *read* access is always possible, although still encapsulated by C++)
Issues	process can corrupt its *own* data; frequently accessed data is moved to shared segments, making it vulnerable after all	bicameral objects (split into protected and non-protected halves); caller must reprotect memory after instantiating or destroying protected objects

Recovering from Faults

❖ Safety Net

❖ Leaky Bucket Counter

❖ Audit

❖ Watchdog

❖ Escalating Restarts

❖ Initialization Framework

❖ Binary Database

❖ Obituaries

Safety Net

❖ base **Thread** class has common entry function

❖ entry function contains a loop that

➢ registers a signal handler that turns fatal signals (e.g. **SIGSEGV**) into exceptions

➢ invokes (**try**) a virtual **Enter** method, which is never exited unless an exception occurs

➢ intercepts (**catch**) all exceptions

➢ on exception, logs stack for debugging purposes and invokes virtual **Recover** function to clean up the running context (e.g. a state machine), after which the loop is reentered

❖ the details, however, can get quite messy

Leaky Bucket Counter

❖ detects when some event has occurred at some threshold frequency (n events in t seconds)

❖ typically used to monitor a fault that does not require corrective action unless it occurs with sufficient regularity

➢ checksum errors on a link eventually cause a reset

➢ traps (signal or exception) in a thread eventually prevent its recreation

➢ traps in a processor eventually cause a restart

Audit

- ❖ background thread that
  - ➢ reclaims orphaned or hung resources
    - ▪ usually based on mark, claim, and sweep strategy
    - ▪ each key resource pool should be audited
      - – object pool audit recovers orphaned blocks
  - ➢ checks consistency of data structures
    - ▪ logs (and perhaps tries to correct) inconsistencies
- ❖ challenges
  - ➢ race conditions that cause transient, false positives
  - ➢ auditing resources whose ownership is distributed
- ❖ initial version should be simple
  - ➢ evolves to fix problems that recur in lab or field

Watchdog

❖ **a sanity timeout that detects a serious failure**

➢ hardware watchdog detects hung system

▪ scheduler or high priority thread periodically resets watchdog

➢ restart timeout ensures that initialization completes

▪ primordial thread sleeps for restart timeout after creating initialization thread

▪ initialization thread wakes primordial thread when finished

▪ if primordial thread times out, initialization took too long

➢ run-to-completion timeout detects thread that runs locked too long (infinite loop)

▪ timeout implemented by **SIGVTALRM** or initialization thread (sleeps for run-to-completion timeout; awoken to cancel timeout when a work thread yields)

Escalating Restarts

❖ strategy for handling fatal errors

➤ too many exceptions; death of a critical thread

❖ return to service quickly and minimize impact

❖ define restart severity levels

➤ warm: restart all threads

➤ cold: warm + reset all non-protected data

➤ reload: cold + reset all protected data

➤ reboot: reload + reboot executable

❖ begin with warm restart and increase severity level if node soon encounters another fatal error

Initialization Framework

❖ **`main`** lacks structure in most systems

   ➢ needs structure to support escalating restarts

❖ each subsystem subclasses from **`Module`** and provides the following functions:

   ➢ **`Startup`**: subsystem "(re-)constructor"

      ▪ e.g. initialize globals, create singletons, launch threads

   ➢ **`Shutdown`**: subsystem "destructor"

      ▪ base restart code tells a subset of threads to exit and destroys a subset of heaps

❖ subsystems register with **`ModuleRegistry`** and specify their dependencies so that the registry can calculate an initialization order

Binary Database

❖ reload restart clears write-protected memory

❖ protected data must therefore be reloaded, which usually takes a long time

➢ for example, database of 100K user profiles

❖ to speed up restart, periodically save a binary image of the database

❖ during restart, reload binary image and reapply any database transactions that occurred since the time the binary snapshot was last taken

Obituaries

- ❖ when an object is deleted during trap recovery, inform collaborating objects

- ❖ when a node restarts, inform collaborating nodes

- ❖ allows objects communicating with the deleted object or restarted node to clean themselves up

- ❖ prevents hung resources or speeds up their recovery

Messaging

❖ Reliable Messaging

❖ TLV Message

❖ Eliminating Copying

❖ Eliminating Messages

❖ Overload Controls

Reliable Messaging

❖ delivery of messages cannot be guaranteed

➢ destination is out of service

➢ destination is unreachable

➢ destination's message buffer is full

➢ destination traps

➢ destination's overload controls discard message

❖ applications must use timers

❖ in most of the above situations, retransmission is pointless or detrimental

➢ applications must not retransmit internal messages

➢ messaging layer retransmits on transient errors

Type-Length-Value Message

Technique	Synopsis
TLV Message	type (parameter identifier) + length + value (contents) — more efficient than text, ASN.1, or XML encodings
Parameter Typing	cast contents as a **struct** to improve reliability and readability
Parameter Fence	**AddParm** places pattern (e.g. **0xaaaaaaaa**) after parameter to detect trampling when next parameter is added
Parameter Template	**AddParm** uses parameter identifier to access a template that initializes the parameter
Parameter Dictionary	parse a message once, when it arrives, and construct a lookup table for fast access to its parameters; table is used by **FindParm**

Eliminating Copying

Technique	Synopsis
In-Place Encapsulation	reserve space at beginning of buffer for prepending header when message travels down protocol stack
Stack Short-Circuiting	bypass lower layers of protocol stack by queueing message directly on destination's work queue
Message Cascading	suppress short-circuiting during broadcasting: `Send` function clones the buffer so the application does not have to reconstruct the message
Message Relaying	apply short-circuiting to an incoming message
Eliminating I/O Stages	ISR->network thread->I/O thread->invoker thread is the usual pattern: move work of I/O thread into network thread, and their combined work into ISR

Eliminating Messages

Technique	Synopsis
Message Attenuation	bundle messages to same destination (a) in I/O thread; (b) at end of transaction; (c) explicitly, in application
Prefer Push to Pull	use Half-Object Plus Protocol to push data changes to other nodes so clients can access data with function calls instead of messages
No Empty Acks	eliminate acknowledgments that do not provide any new data
Polygon Protocol	eliminate intermediate acks by sending final ack to original requester (A req B req C ack B req D ack B ack A becomes A req B req C req D ack A)
Callback	invoke observer's callback function instead of sending it a message [has drawbacks that must be considered]
Shared Memory	move frequently accessed data to a shared segment

Overload Controls

❖ system receives *n+m* jobs but can only handle *n*

➢ typical system does *n+m* jobs poorly: thrashes or crashes

➢ extreme system rejects work to crest at *n* jobs asymptotically

Technique	Synopsis
Finish What You Start	handle progress work before servicing new work
Discard New Work	limit length of new work queue to prevent resource gobbling; throw away new work that was queued for longer than some threshold; match cancellation with request and discard both
Ignore Babbling Idiots	ignore or reset a client that sends more than *n* messages in *t* seconds (detected by Leaky Bucket Counter)
Throttle New Work	send new work credits to clients; stop sending credits if overloaded (detected by CPU occupancy, delay in new work queue, or length of new work queue)

Failover

- ❖ Failover Strategies: Overview
- ❖ Failover Strategies: Comparison
- ❖ Checkpointing

Failover Strategies: Description

Technique	Synopsis
Load Sharing	➢ group of processors share workload ➢ remaining processors take over all work if one fails
Cold Standby	➢ inactive (spare) unit substitutes for failed unit ➢ also known as "n+m sparing"
Warm Standby	➢ two processors run independently ➢ active unit handles I/O and work, checkpointing data to the standby unit to keep it synchronized ➢ standby unit takes over if active unit fails
Hot Standby	➢ two processors run in synch ➢ both receive messages; only active unit sends ➢ on disagreement, processors decide who is sane ▪ sane unit takes (or retains) control ▪ faulty unit removed from service and diagnosed

Failover Strategies: Comparison

	Load Sharing	Cold Standby	Warm Standby	Hot Standby
Hardware cost	n+m; off-the-shelf	n+m; off-the-shelf	2n; off-the-shelf	2n; customized
Hardware fault transparency	no work survives	no work survives	checkpointed work survives	all work survives
Recovery time	fast	slow (often requires download)	fast	very fast
Capacity impact	n/(n+m) to allow takeover	n/(n+m) to allow sparing	reduced by checkpointing	reduced by h/w synching
Checkpointing logic in applications	no	no	yes	no—but still needed for hitless upgrades
Software fault transparency	excellent	excellent	excellent	synchronized insanity
Typical usage	stateless servers	low-end servers	high-end stateful servers	high-end servers

Checkpointing

Technique	Synopsis
Application Checkpointing	➢ applications perform *ad hoc* checkpointing ➢ bulk checkpointing through application callbacks
Object Checkpointing	➢ objects added to registry as data changes ➢ registry builds message at end of transaction by calling `Pack` function to serialize each object ➢ inactive unit calls `Unpack` functions when message arrives ➢ class and object identifiers replace pointers
Memory Checkpointing	➢ dirtied pages checkpointed at end of transaction ➢ requires identical memory layouts
Virtual Synchrony	➢ simulates hot standby in software ➢ inputs go to both units, which run in parallel ➢ unit is resynchronized if it fails to respond ➢ timers and bulk checkpointing are problematic

Installing Software

- ❖ Hitless Patching
- ❖ Hitless Upgrade
- ❖ Rolling Upgrade

Hitless Patching

❖ no interface changes allowed

➢ cannot change layout of data structures

➢ cannot add or override functions

➢ to support patching, `Object` class defines

▪ `void* patchData_` to add data housed in a block

▪ `virtual Patch(int selector, void* args)` to add functions; subclass invokes superclass `Patch`

❖ function is patched by

➢ modifying and compiling it

➢ inserting the object code into a running system

➢ modifying the original code to jump to the new code

❖ need patch administration system

Hitless Upgrade

1. take CPU1 (standby) out of service
2. insert new release in CPU1
3. load CPU1 with configuration data
   - data must be reformatted to layout in new release
   - binary database can be created in advance
4. put CPU1 back in service
5. wait for CPU0 (active) to resynchronize CPU1
   - checkpointing between different releases
6. manually force a failover to CPU1
7. repeat steps 1-5 to upgrade CPU0
   - only after new release has run without incident

Rolling Upgrade

❖ upgrade nodes one at a time

➢ avoid total outage or flash cutover to unproven release

❖ order of rolling upgrade is

1. administration node (for new configuration data)
2. control node (to talk to administration node)
3. access nodes (simpler than service nodes)
4. service nodes

❖ requires backward compatibility of protocols

➢ some nodes running new release, others old one

➢ new release reformats messages to/from old one

➢ support upgrade to release n from n-2 or n-3?

Software Optionality

❖ Reasons for Optionality

❖ Techniques for Optionality

Reasons for Optionality

❖ product families

➢ share software, not all of which is needed

❖ multiple platforms

➢ abstraction layer hides differences

❖ optional features

➢ disable capability until customer pays for it

❖ field debugging tools

➢ disabled until required

Techniques for Optionality

Technique	Synopsis
Conditional Compilation	➢ **#ifdef** grossly overused in most systems ➢ restrict use to supporting different platforms by *wholly* including or excluding .cpp's that implement a platform adaptation layer defined by a common set of headers
Software Targeting	➢ use makefiles or other build tools to select a subset of directories ➢ best way to exclude entire subsystems from specific products
Run-Time Flags	➢ check flags (default to **false**) at run-time ➢ need interface for enabling flags ➢ best way to support optional features and field debugging tools

Debugging

❖ Debugging Requirements

❖ Logs for Debugging

❖ Trace Tools for Debugging

Debugging Requirements

❖ **must be safe enough for field use**

➢ no service disruptions; minimal capacity impact

▪ cannot set breakpoints or write to console

▪ must capture information selectively

❖ **must usually be good enough for lab use**

➢ no other techniques will be available in the field

Logs for Debugging

Technique	Synopsis
Software Error Log	➤ associated with safety net ➤ unexpected (signal) or deliberate (application detects serious error and throws exception) ➤ provides full stack trace (functions, arguments, local variables)
Software Warning Log	➤ deliberate (application detects non-fatal error) ➤ application provides reason/value and offset
Object Dump	➤ symbolic display of object recovered by safety net or audit
Flight Recorder	➤ history of software logs, alarms, restarts, failovers, interventions by administrators ➤ saved in memory that survives restarts ➤ used to debug outages

Trace Tools for Debugging

Technique	Synopsis
Function Tracer	non-trivial function calls a trace function that records the function's name, its stack depth, the running thread, and the current tick time; postprocessor provides an indented display of function calls and timing information
Message Tracer	captures contents of incoming and outgoing messages; postprocessor provides symbolic display of contents
Tracepoint Debugger	captures memory contents (specified by interpreted opcodes) at debug tracepoints; immediately resumes execution

❖ **tracers share a large buffer and support triggers to limit the amount of information captured**

➢ trace specific threads, functions, protocols, signals, IP addresses, IP ports…

Two Programming Models

Common	Extreme
heap	object pools
foreground garbage collection	background object pool audit
preemptive scheduling	cooperative scheduling
synchronous messaging	asynchronous messaging
many short-lived threads	fewer threads; almost all are daemons
priority scheduling	proportional scheduling
user spaces	write-protected memory
trap causes reboot	safety net
thrashing	overload controls
shutdowns for upgrades	in-service patching and upgrades
unconditional breakpoint and `printf` debugging	conditional tracepoint debugging and trace tools designed for live systems