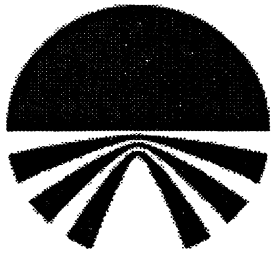


Bay Networks

GAME & Kernel Overview

10/24/96

GAME Background

GAME = Gate Access Management Entity (acronym probably came first).

Original Wellfleet ACE system was a VME architecture with VRTX based software.

This had a few shortcomings:

- **Performance - everything wound up in a single process so we weren't really using the OS.**
- **Hotswap - HW could do it. SW couldn't.**
- **Dynamic reconfiguration wasn't possible.**
- **Hard to support - no source code for the OS.**

GAME developed to address deficiencies with the ACE systems.

GAME was developed in conjunction with new HW - FRE.

How do you build a router?

Before discussing GAME proper its best to look at some of the design considerations which need to be addressed in order to build a good multi-protocol router.

A router is much different than a 'normal' PC or Unix workstation design.

Some of the design issues include:

- **Encapsulation/decapsulation.**
- **Performance.**
- **Packet access.**
- **Multi-slot issues.**
- **High availability.**
- **Dynamic reconfiguration.**
- **Internal code structure.**

Encapsulation

Encapsulation/decapsulation must be efficient.

This is one of the main jobs of a router -- removing one type of header and adding another.

Copying data is bad -- very inefficient.

Note: you may need to add more data than is removed.

Solution: Provide extra pad space before packet data so data can be appended without copying.

Performance

One way to higher performance is to effectively utilize the caches:

Instruction cache:

- **Traditional locality of execution.**
- **But, too much code leads to a point of diminishing return as I-cache footprint gets overwritten (initially this was 1991 technology).**

Data cache:

- **Two hot areas: stack and routing table.**
- **Routing table very important.**

Solution: Keep amount of code being executed to a reasonable size.

Operate on more than one packet in a row so to utilize I-cache and routing table in D-cache.

Packet Access

Packet access is different from normal data access because other hardware (link module, PPX) share the memory.

Hard to make this cacheable especially if processor doesn't have the hooks (MESI).

Could perform cache flushes/invalidates before giving packet to other HW, but...

Observation: CPU doesn't care about most of the packet. Only really cares about the headers, not the data. Also, the headers aren't accessed multiple times.

Implies caching wouldn't be such a big win.

Solution: Keep packet memory non-cached.

Provide special hardware to accelerate accesses to hot spots (packet headers).

Multi-slot

Packets may come in a interface on one slot but go out an interface on another slot.

Should the forwarding path really care about this?

Ideally no. It'd be nice to perform one lookup and just mark where the packet goes next without regard for whether or not it a remote slot.

This would also isolate local vs. remote decision to one piece of SW.

Solution: Provide an addressing mechanism where local and remote are treated equally.

High Availability

High availability = no single point of failure in the box.

This implies that tightly coupled slots (shared memory) are bad or one slot could bring down another.

But, still need to know the state of other slots so as to know if remote interfaces are still working.

How about SW? Want to prevent an error in one protocol from affecting other protocols.

Solution: Communicate slot to slot with messages that don't use shared memory.

Provide a mechanism to monitor remote slots.

Track SW resources so when a protocol crashes its resources can be reclaimed and the protocol restarted.

Dynamic Reconfig.

Much like high availability.

Need to be able to remove a protocol without affecting other protocols.

Solution: Track resources (same as high availability).

Internal Code Structure

Could have one big function for forwarding. But,

- **Not very maintainable.**
- **Hard to change processing steps (which may be configuration dependent).**
- **Too big and you ruin your I-cache footprint.**

How about error recovery/reconfiguration?

- **Need a way to know what code is related so all pieces are removed.**
- **Resource tracking must be enforced.**

Solution: Arrange code into manageable sized 'processes'.

Arrange in parent/child fashion to make it easy to group related gates.

Design Summary

Design consideration	GAME Solution
Encapsulation	Buffer format where data is 'suspended' below a header.
Performance	Gate design where code for a single routing 'step' is done. Operate on lists of packets. Include the destination for the buffer within the buffer.
Packet access	HW acceleration for packet headers.
Multi-slot	Gatehandles which represent local/remote in same format. OS interprets if destination is local or remote.
High Availability	Mappings to track existence of gates (local or remote). OS tracks all resources owned by gates. Gates arranged in parent/child.
Dynamic reconfig.	Same as High Availability.
Internal structure	Gates are lightweight. Buffers contain their own next destination. Dynamic loader.

Gates

A Gate is the fundamental unit if scheduling.

Analogous to a process, task or thread of other OSs.

Every gate has an ID, referred to as a GID.

There are 5 types of gates:

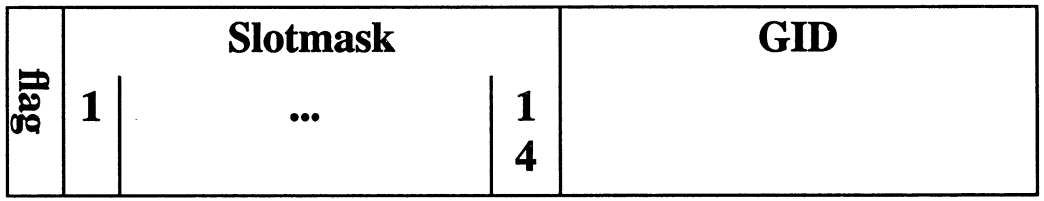
1. **Well-known -- Can exist on multiple slots. GID is defined at compile time.**
2. **Dynamic -- Exists only on a single slot. GID is allocated at run time.**
3. **Alias -- A GID which groups gates. Any message sent to an alias has a copy made for all the gates in the alias list.**
4. **Ensign -- A GID with no gate block. Can only be mapped.**
5. **Davidian -- A GID with no gate block. Can only be mapped on same slot.**

Gates are related to one another in a parent-child relationship.

A gate can be a "soloist". A well-know gate which only exists on one slot but which will move if that slot dies.

Gates continued...

The existence of a gate is tracked with a gatehandle. This is a GID along with a slot-mask which tells which slots that GID exists on:



Each gate has a activation function which can be changed while executing.

Each gate can malloc its own environment or have its parent provide one.

Children are allowed to access their parents memory. The reverse, parents accessing a child's memory, or accessing memory of a different family tree is highly discouraged because of high availability considerations.

Gate activation is just calling its activation function passing the gate's environment and a buffer list or signal as arguments.

Many gates (1000s) can be supported.

The first gate created is the loader gate. This is done by the kernel. All other gates are then children of the loader.

Mappings

Mappings are (probably) unique to GAME.

Allow any gate to track the existence of any other gate in the entire system.

Tracking is done via gatehandles.

Uses:

- **Find out where protocols are loaded -- map IP's well-known gate.**
- **When forwarding traffic, make sure the interface is up -- map the circuit gate.**
- **Keeping an eye on children -- map them.**
- **Cleanup in case of error -- map yourself.**
- **Notification of MIB changes -- map a davidian created by the MIB for you.**

Mapping run as a temporary child of the gate which created it.

A mapping activation gets the old gatehandle and the current gatehandle and compares the two to see what has changed.

A mapping activation is called for each gatehandle change.

Scheduler

Basic scheduling is a FIFO -- first in / first out.

Gates run to completion or until they make a pending system call.

All gates share the same stack unless they pend at which point the current stack is saved and a new stack allocated.

The scheduler has 2 queues:

- **Active queue -- list of gates ready to run.**
- **Idle queue -- list of gates ready to run after scheduler goes idle.**

A gate is activated for only one reason at a time. If a gate pends it will not be activated for such things as delivery of new buffers until after it returns from its first activation.

If scheduled with a list of buffers, the gate is expected to do something with those buffers (send them somewhere else, free them, add to private pool).

Long running gates are expected to give up the CPU to allow others to run.

Interrupts

Interrupts fall into 3 different classes:

1. **Always enabled -- such things as the watchdog, error detection and tty breaks.**
2. **Enabled between gates -- timer, backbone.**
3. **Enabled when scheduler idle -- module.**

Each class also includes the interrupts from the class above it.

Interrupts such as timer and backbone are only taken between gates for performance reasons. This allows forwarding gates, which are processing many buffers, to execute uncontested.

Module interrupts are take only when idle in order to allow the hardware time to build up a list of buffers for processing. This is important as lists are much more efficient to process then individual buffers.

Signals

There are a few different types of signaling mechanisms:

SIG_INI:

- **The creator of a gate can arrange for that gate to get an initialization signal.**

g_sig/g_isr:

- **A gate registers to handle a signal via g_isr. Some other entity "throws" the signal via g_sig.**
- **The thrower doesn't know who the signal catcher is (or even if one exists).**
- **This is how GAME lets drivers know when a HW interrupt occurred.**
- **Multiple g_sig's which occur before the gate is activated for that signal do not nest.**

g_sig_data:

- **Allows a gate to send multiple signals to another gate.**
- **Can optionally send memory along with the signal.**
- **Designed to get around a early GAME limitation of only allowing one user defined signal per gate.**
- **Multiple g_sig_data signals can be sent to a gate and none will be lost.**

Timers

GAME provides a simple timer facility.

Each gate is allowed to have one timer.

Gates can arrange to receive a periodic timeout signal which will result in a SIG_TMO scheduling.

Timers automatically reset until canceled.

There is a delay mechanism whereby a gate can pend in place for a specified amount of time.

A word of warning: accurate timers were never a design goal of GAME and thus they aren't very accurate.

A few reasons why:

- **Timer interrrupts are only handled between gates thus if a gate runs through multiple timer ticks GAME will miss them.**
- **Gates get scheduled at the end of the activation queue and need to wait for others in front of them to complete.**

Bay Networks

Memory Model

GAME operates with a flat memory map (logical = physical).

There are 3 basic memory types:

Malloc memory:

- **"Normal" memory used for mallocs, stacks and loadable applications.**
- **Cacheable.**

Buffer memory:

- **Special memory used exclusively for buffers.**
- **Buffers are carved at initialization time and there is a fixed number of them.**
- **Shared between processor, backbone and link so not cacheable.**
- **Often has special hardware to improve its performance.**

HW registers:

- **All accesses to HW is via memory mapped registers.**
- **Not cached.**

Buffers

Buffers are carved at initialization time.

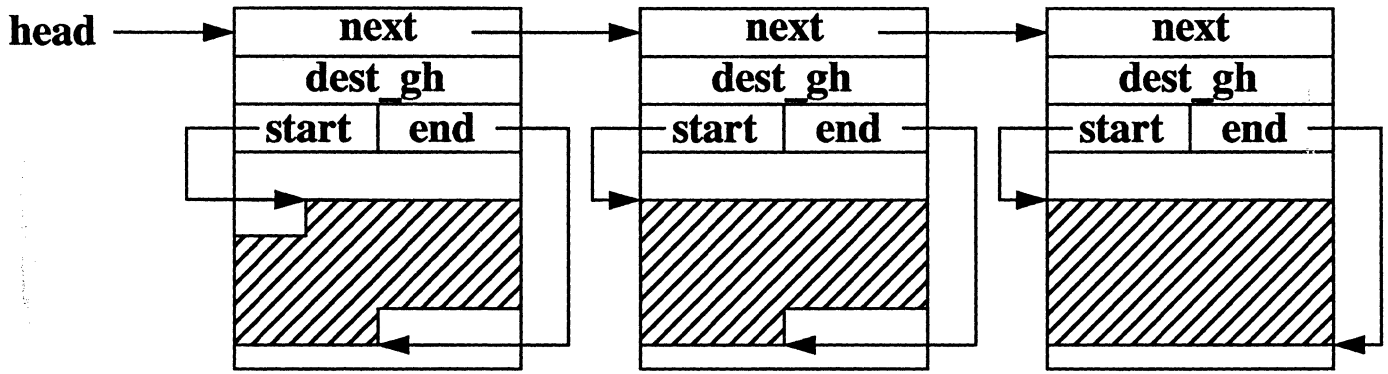
A buffer is always on exactly one list.

There are numerous lists throughout the system:

- **Free buffer list.**
- **Each gate can be delivered a list and can hold onto a list in its "private pool".**
- **The PPX transmitter and receiver can own lists.**

The data in a buffer is in the middle of the buffer to allow for easy encapsulation.

A buffer also contains a destination gatehandle used by the message delivery system.



Messaging

GAME provides two types of messaging; unreliable and reliable.

Unreliable messaging:

- **This is the normal type of message used in the forwarding path.**
- **These messages may be lost in the system, usually because the receiving slot has no more receive buffers.**
- **Gates can send lists of messages this way (lists are much preferred).**
- **Sending unreliable messages does not pend the gate.**

Reliable messages:

- **Primarily used by the control path.**
- **GAME uses an acknowledgment protocol to ensure delivery to remote slots.**
- **Reliable messaging will pend the gate until the message has been delivered (or an error occurs).**
- **There is an RPC mechanism where a gate can send a message and pend until the replies have been collected.**

Semaphores

GAME provides a semaphore mechanism.

Semaphores can be well-know (compile time) or dynamic (run time).

Gates must register for semaphore usage before obtaining a semaphore token. This allows GAME to remove a semaphore should all its users cease to exist.

The number of tokens in a semaphore can be varied.

There is a way to check for token availability before trying to get the token.

SMP

Some new hardware platforms support symmetric multiprocessing.

GAME's implementation of SMP makes use of the family tree and gate classification to determine which gates may run in parallel.

Gates are classified depending upon what other gates they share memory with.

This SMP model was developed to be backward compatible with existing code where the gate knows it owns the CPU until it gives it up and thus can modify memory at will.

The goal is to have forwarding path "clean" so it can run in parallel with any other gate.

High Availability

In order to obtain high availability GAME track all resources owned by a gate:

- **Buffers**
- **Memory**
- **Children**
- **Presence in activation queue.**
- **Ownership of semaphore tokens.**

If a gate dies due to an error, it and all its children are removed and all their resources are freed.

Mappings are then triggered to notify other gates of the change.

Re-configuration uses this same mechanism to remove an actively running protocol.

GAME also tracks resource usage by a gate and may log an error or kill the gate if it:

- **Uses lots of memory.**
- **Pends while owning many buffers.**
- **Hogs the CPU for a long amount of time.**

Other GAME Services

GAME provides an event log:

- **This is a special region of memory which will survive a crash intact. About the only time its lost is when the board is power cycled.**
- **Gates can log information here; warnings, faults, debug info, etc.**
- **Can be viewed from the TI, Optivity, saved to flash or tftp off the router.**
- **Provides postmortem crash analysis capabilities.**

There are system calls to read and set the wallclock time.

There is a way to check the platform type to make runtime decisions (though these are rare).

Bay Networks

Other Parts of the Kernel

GAME usually refers to the OS while 'kernel' often includes other subsystems which are build on top of GAME.

The kernel is the statically linked piece, it doesn't include applications which are dynamically loaded.

Subsystems in the kernel include:

- **Dynamic loader -- Loads applications (i.e. IP).**
- **MIB -- The main database though which the router is configured and statistics retrieved.**
- **Filesystem -- Provides access to flash cards.**
- **DP (datapath) -- Provides decapsulation and encapsulation of MAC layer.**
- **TBL -- Provides utilities for fast table lookups.**
- **TI (technician interface) -- Provides rudimentary console interface.**

Loadable Applications

Dynamically loadable applications use special compiler flags so that the code generated can run at any address without the help of a MMU.

The loader will attempt to obtain applications directly from another slot's memory before resorting to loading off the flash.

This leads to a few restrictions:

- **Apps shouldn't have global data. Everything should be kept in the environment.**
- **Pointers within the .data section can be problematic as they are not automatically relocated. The app needs to relocate these itself.**
- **Apps need to realize that other apps can come and go, taking their services with them. An app should map the root gate of any other app it makes use of.**

Simple Packet Flow