GAME Overview

Bill Miller

Wellfleet Communications Inc.

10/24/94

GAME OVERVIEW

- **Historical Perspective**
- **Founding Principles**
- **Hierarchical Model**

HISTORICAL PERSPECTIVE

ACE PLATFORM

- **FN/LN/CN**
- **Motorola 68K (ACE25, ACE32)**
- **VME-based Interconnect**
- **VRTX Realtime OS**
- **Performance up to 20K pps**

BACKBONE NODE PLATFORM

- **BLN/BCN**
- **Motorola 68K (FRE-I, FRE-II)**
- **Proprietary Backbone Interconnect**
- **Proprietary Real-time OS (GAME)**
- **Performance up to 100K pps**

FOUNDING PRINCIPLES

MANAGEABILITY

- **Dynamic Reconfiguration vs. Total Reboot**
- **Hot Swappable Motherboards, Link Modules, Power Supplies, etc...**
- **Fault Resiliency - No single point of failure, Isolate crashes, Distributed vs. Soloist**
- **Remote User Interface (Site Manager) - remove ASCII text and code from router**

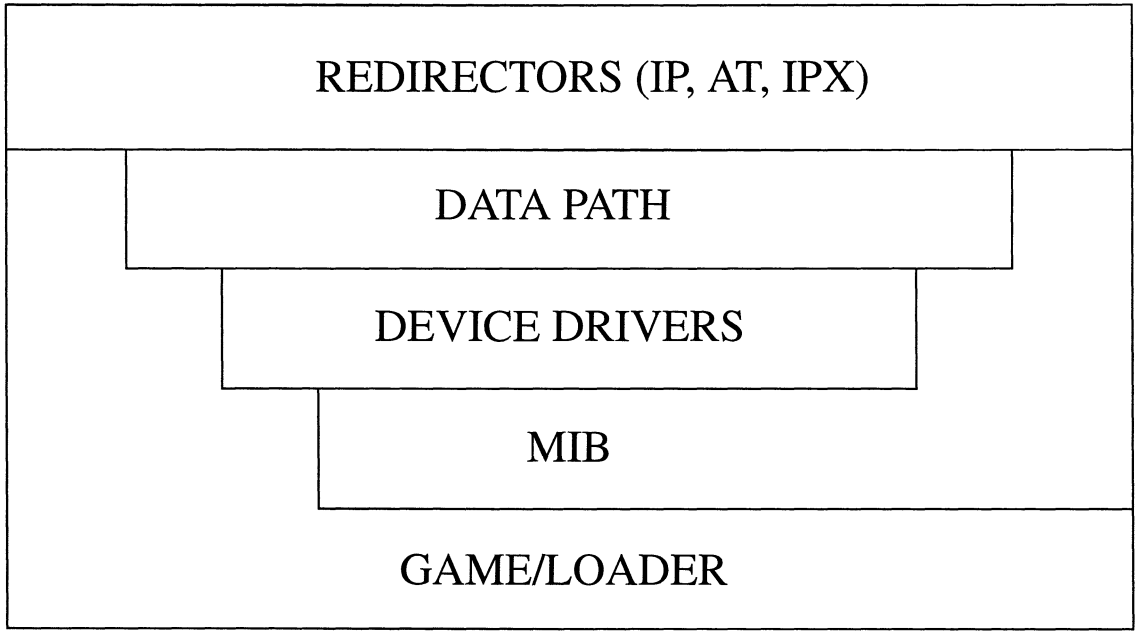
PERFORMANCE

- **GAME - designed to pass buffers efficiently, fast context switches**
- **Max Headroom - peform decaps/encaps on source/destination slots**
- **Memory Reclamation vs. Dynamic Loading - improved memory utilization**
- **Floppy vs. Flash File System - faster boot times, file transfers**

DEVELOPMENT

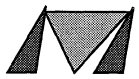
- **Powerful simulation environment running on workstations - time/cost savings**
- **Portability of Kernel to various platforms**
- **Debugging hooks for developers... "The Log"**

HIERARCHICAL MODEL



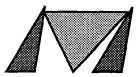
Topics

- **What is Game?**
- **Important data structures.**
- **Gates and their environment.**
- **Messaging.**
- **Mappings.**
- **Kernel Services:**
  - **Gate Management**
  - **Buffer Management**
  - **Inter-gate Communication**
  - **Time Services**
  - **System Logger**
  - **Semaphores**
  - **Other Utilities**



What is GAME?

- **GAME = Gate Access Management Entity.**
- **A gate is analagous to a process in most other operating systems.**
- **Multi-process - many 1,000's.**
- **Multi-processor - communicates via messaging with other slots.**
- **Non-premptive - gates run to completion or until they give up the CPU.**
- **Round robin scheduling - no gate priority (except for mappings).**
- **Support for hardware and software fault management.**
- **Support for dynamic reconfiguration.**

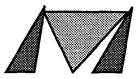


Data Structures - Gatehandle

- **Gatehandle is composed of two parts: slot mask and gate ID.**
- **Gate ID (GID) identifies specific gates.**
- **Slot mask identifies which slot that gate is on.**
- **Different slots may contain gates with the same GID - these are "well-known" gates.**

Data Structures - Buffers

- **Buffers are the basic messaging structure for the system.**
- **All buffers are carved and created at init time.**
- **Buffers are singly linked into lists.**
- **A buffer is always on exactly one list.**
- **The BackBone hardware understands buffers and will walk buffer lists when sending or recieving packets off of a rail.**
- **Buffers can be marked "reliable" or "unreliable".**
  - **Reliable buffers are used mainly in the control path.**
  - **Unreliable buffers are used for data.**



Data Structures - Buffers

Buffer Chaining

Gates and their Environment

- **Each gate is created by another gate in a parent/child relationship.**
  - **The root gate of a driver or application is created by the loader gate (which is one of the few gates created directly by GAME).**
- **Each gate has a state: dormant, awake, active, pended or zombie.**
- **Each gate has an activation routine which is the function the scheduler calls to run the gate.**
- **Each gate has an environment which is typically memory malloc'd by its parent or by itself.**
- **GAME tracks all resources (buffers, memory, signals, etc.) owned by a gate and will free these resources if a gate dies (or is killed).**
- **When a parent dies, all its children die.**
- **If a child dies, nothing happens to the parent.**
- **Sharing memory between parent and its children is ok. But, not from child to parent or between unrelated gates. This is because GAME will free that memory if the gate dies.**

Gate Creation

- **Gates are created with the g_req() syscall:**

```
g_id = g_req(gid, act, env, opt)

   GID gid          GID to modify.
   void (*act)()    Activation routine.
   u_int32 env      User defined environment.
   u_int opt        Initialization option.
```

  - **When creating a new gate the `gid` may be either a well-known GID or `G_REQ_NEW_GID`.**
  - **g_req() can be used with a gid of `G_SELF_ID` to change act or env.**
  - **An activation of `G_REQ_KILL` will kill a gate.**
  - **env is just a number from GAME's point-of-view. Usually its a pointer to a gate's environment.**
  - **The options are:**

G_SIG_INI	Causes new gate to be scheduled with SIG_INI.
G_REQ_INI	Causes existing gate to be scheduled with SIG_INI.
G_REQ_SOLO	Requests a soloist election- well-known gates only.

- **The new gate becomes a child of the current gate, which is the parent.**

Gate Activation

- **Gates are activated for either buffer delivery or signal delivery - never both.**
- **Once activated, that gate's activation routine will not be called again until the current activation has completed.**
- **A gate must process all the buffers on its delivery list before returning. Faliure to do so is an "orphaned buffer" error.**
- **The activation routine will run to completion or until the gate pends.**

```
void gate_action(env, buf, sig)

   u_int32 env       Environment set with g_req.
   BUF *buf          List of buffers being delivered.
   SIG signal        Signal being delivered. Valid only if buf = 0.
```

Gate Sample Code

```
/* Example code - This is a "parent" gate which sets up its environment,
 * and prepares to handle buffer delivery. It was created with:
 *
 * g_req(G_REQ_NEW_GID, my_init_act, 0, G_SIG_INI);
 */

void my_init_act(u_int32 env, BUF *buf, SIG sig)
{
   /* First, we _have_ to check for buffers. */
   if(buf)
   {
      PANIC;      /* We don't want any yet. */
   }
   else if(sig == SIG_INI)
   {
      env = g_malloc( /* some env size */ );

      /* Now change my env and activation */
      g_req(G_SELF_ID, my_buf_act, env, 0);
   }
}


void my_buf_act(u_int32 env, BUF *buf, SIG sig)
{
   if(buf)
      my_handle_bufs(buf);
 etc....
```

Messaging

- **Messaging can be either reliable or un-reliable.**
- **All traffic on the backpanel is unreliable.**
- **GAME effects reliable delivery with an acknowledgement mechanism for reliable packets.**
  - **GAME will also retry reliable transmission.**
  - **Reliable buffers have a sequence number and source GH.**
  - **GAME eliminates old replies via messaging window.**

Mappings

- **Allows for tracking of any GID in the system.**
- **The mapping function is run whenever the GH of the mapped GID changes state.**
- **Mappings are created with the g_map() syscall:**

```
void g_map(gid, gh, act)

   GID gid          GID of gate being mapped.
   GH *gh           Pointer to gate handle in caller's space.
   void (*act)()    Activation routine.
```

- ***gh maintains the previous state.**
- **Mappings activated with the old and new gatehandles:**

```
(act *)(gh, new_gh)

   GH *gh          The *gh from the g_map() call which contians the old state.
   GH new_gh       The current state.
```

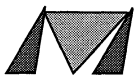
- **Mappings are run in a new, temporary, gate at the <u>head</u> of the scheduler queue.**
- **A mapping's environment is the creator's environment.**
- **Mappings are independent of the creator's main activation routine and thus can change the creator's environment while the creator is pended or asleep.**
- **A gate can map itself with a gid of G_SELF_ID.**

Mappings Cont.

- **When gates are dying, the order of events is:**

  1. Fire self mappings of dying gates.
  2. Remove dying gates (free resources).
  3. Fire all other mappings of dying gates.

- **Mappings are removed with an act of G_UNMAP.**
- **If a mapping is unmapping itself then the g_map(... , G_UNMAP) call does <u>not</u> return.**



Mapping Sample Code

```
/* Someplace in an activation routine... */

g_map(G_SELF_ID, g_malloc(sizeof(GH)), my_map_act);
```

```
void my_map_act(GH *gh, GH new_gh)
{
    if(GH_BECAME_LOCAL(*gh, new_gh))
    {
        /* Typically, I don't care when I became alive.  Just update *gh
         * and exit.
         */
        *gh = new_gh;
        return;
    }
    else if(GH_CEASED_LOCAL(*gh, new_gh))
    {
        /* I'm dying.. do some cleanup... */
    }
}
```

Gate Management

g_id = g_req(gid, act, env, opt)

- **Creates, modifies and kills gates.**

g_zona(fname, line)

- **Used by the PANIC macro.**

alias_gid = g_alias(alias, gid, opt)

- **Allows gates to be grouped into sets. Buffers can be sent to the alias_id and each gate in the group will get its own copy.**

g_map(gid, gh, act)

- **Creates or removes a mapping.**

g_isr(gid, signal, flag)

- **Registers a gate to handle a specific signal.**

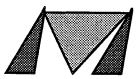
g_myid()

- **Returns the callers GID.**

Gate Management (pt. 2)

env = g_env()

- **Returns the gates env.**
- **Useful for a mapping to retrieve the creator's env.**



Memory Management

mseg = g_malloc(size)

- **Mallocs a new memory segment from the free memory pool.**

g_mfree(mseg)

- **The gate must own the memory in order to be able to free it.**

llen = g_mlen()

- **Returns the largest memory segment. Usually, a g_mlen() is called before a g_malloc() to ensure that the g_malloc() will succeed.**

g_madd(new, size)

- **Adds a new segment - usually done by powerup code.**

Buffer Management

buf = g_balloc(tmo)

- **Allocates a buffer with a possible timeout.**

cnt = g_breplen(num, head, tail)

- **Allocates a list of buffers. Usually used by the drivers.**

g_bfree(head, tail)

- **Free's a buffer list.**

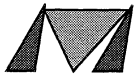
g_bsave(head, tail)...

g_brestore(head, tail)...

buf = g_bhead()...

buf = g_btail()...

- **Manipulates buffers on a gate's private pool.**



Buffer Management (pt. 2)

len = g_blen()

- **Returns the maximum buffer end offset for this slot. G_BUF_MAX_END is the system's max end offset.**

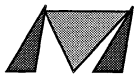
g_bmove(insert, head, tail)

- **Moves buffers around on the transient list.**

g_bsave_dbl(head, tail)

len = g_brestore_dbl(map, arrayp)

- **Manipulates doubly linked privete buffer pool.**



Inter-gate Communication

g_xmt(buf)

- **Sends a buffer list unreliably.**

g_fedex(dest_gh, head, tail)

- **Quickly sends an unreliable buffer list to a local gate.**

slot_map = g_fwd(gh, buf)

- **Send a reliable message.**

reply * = g_rpc(gh, buf)

- **Performs an rpc.**

g_reply(buf)

- **Replies to an RPC. Caller shouldn't modify start offset.**

fwd_id = g_fwd_list(dest_gh, head, tail, pipe_id)

- **Allows for creation of a reliable "pipe" between gates. Messages are sent with g_fwd().**

src = g_src(buf)

- **Gets the source of a g_fwd'd message.**

Inter-gate Communication (pt. 2)

g_repeat(act, env, buf)

- **Calls act with single buffers.**

copy = g_copy(buf)

- **Copies a buffer.**

g_sig(signal)

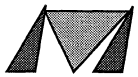
- **Generates a software signal. The gate registered with g_isr() will be activated.**

status = g_sig_data(dest_gid, type, data)

- **Allows sending an arbitrary signal and a memory segment to another local gate. That gate gets scheduled with a SIG_DATA.**

status = g_get_sig_data(*sender, *type, *data, *size)

- **Call made after recieving a SIG_DATA to get the important info.**



Time Service

g_tmo(gid, tmo)

- **Sets or cancels a signal. Signals are not reliable (time wise). Gate gets scheduled with a SIG_TMO.**

g_tget(*tb)

- **Gets the current time.**

g_tset(*tb)

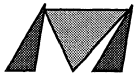
- **Sets the current time.**

g_idle(opt)

- **Idles the caller. Can either go to the end of scheduler queue or onto pending queue.**

g_delay(tmo)

- **Delay for a while, giving up the CPU for some amount of time.**



System Logger

g_log(event, args...)

- **Adds an event to the error log.**

g_stk(level, opt, time)

- **Saves the current stack in the log.**

buf_dump(buf)

- **Dumps a buffer to the log.**

Semaphores

id = g_sema(sema, n)

- **Creates or registers for semaphore usage.**

g_sema_get(sema)

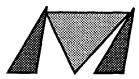
- **Gets a token.**

g_sema_put(sema)

- **Puts a token.**

num = g_sema_state(sema)

- **Returns number of free tokens or waiters.**



Utilities

slot = g_slot()

- **Returns this slot's number.**

type = g_platform()

- **Returns the patform type.**

base = g_appbase(name)

- **Gets base address of application. Needed for dynamic tables.**

g_reset(gh)

- **Resets slots**