HW2

Answer the following questions. For questions asking for short answers, there may not necessarily be a "right" answer, although some answers may be more compelling and/or much easier to justify. But I am interested in your explanation (the "why") as much as the answer itself. Also, do not use shorthand: write your answers using complete sentences.

(GMS Design) In the GMS paper, only "clean" pages are written to global memory ("dirty" pages must be written to disk before they can participate in the GMS pool).

1. Why did GMS chose this restriction? What problem does it solve? It is to prevent data loss, as clean pages will always be available in disk, even when requesting the memory, the corresponded node is down, it can still be recovered from the disk. As system only deal with clean pages, higher level software is responsible for managing the shared memory for sync. It provides the same guarantee as NFS.

2. Describe how you might change GMS to safely allow it to write dirty pages to the global memory. It may be possible that we populate/broadcast changes of dirty pages to all nodes so that unless the whole cluster is down, we won't loss data. However, this action will be pretty expensive.

3. Why does GMS require trusted nodes? What risks would be exposed by an untrusted node? The whole algorithm is based on the fact that all nodes are trusted but may be down at anytime. If any of the node is acting differently, they could:

- Lower system's overall performance by forcing other nodes to evict more memory and request other node to store data for it
- Corrupt global memory integrity by sending invalid data to requester

4. Describe how you might change GMS to work with untrusted nodes? We may need to have a central server to keep track each global memory state and all requests has to be validated by this central server. This server may be hard to scale and be the bottleneck when the cluster become larger and larger.

(Transparency vs Optimization) Butler Lampson once gave a set of principles for system design. Among these, he gave two conflicting pieces of advice on the nature of implementations. He said,

> "Keep secrets of the implementation. Secrets are assumptions about an implementation that client programs are not allowed to make...

> Obviously, it is easier to program and modify a system if its parts make fewer assumptions about each other."

And yet,

> "One way to improve performance is to increase the number of assumptions that one part of a system makes about another; the additional assumptions often allow less work to be done, sometimes a lot less."

That is, on the one hand we should hide an implementation for ease of development (transparency), and, on the other, we should expose our implementations for speed (optimization). Consider this issue for each of the following three systems – Sprite, Xen, Grapevine. For each System

1. Which advice of Lampson's did the authors follow? Describe the service that was implemented, and whether the authors chose to hide or expose in the implementation.
2. Describe what was hidden or exposed in the implementation, and the software mechanisms that were used to do the hiding or exposing. Be specific.
3. Give a concrete example of how the mechanisms above were used to hide or expose in the system.
4. Describe one problem the authors had in utilizing their mechanism for their respective purpose, and how the authors dealt with that problem. Be specific.
5. Given the quotes above, discuss the authors' goals in following the design principle they chose. Did they achieve them? Justify your answer.

Sprite

1. Sprite tried to hide the implementation of the distribution (name and location), making users feel like they are controlling a single computer where underneath there is a cluster of them.
2. There are several of them:
    1. File system: when user accesses the file, it feels like the files are on the local machine, where underneath it, system use prefix table to locate which machine has the file and fulfill the request
    2. Process migration: System might migrate process to idle machines, while on the user side they won't realize it. Still, some local related calls will be redirect to the original machine to complete, and when the remote machine is busy, the process might be moved to different location again

    3. Cache: cache is used to reduce delays to remote disk access, and server side cache may be larger to provide better throughput to all users

3. When hiding distributed file system nature, each computer will have their own prefix table, which uses to determine if a file is locally available, or a RPC is required to be send to the target machine. At the same time, when a machine found the corresponded file, it will broadcast to all machine to invalidate the entry on the prefix table

4. One issue will be at the time where two files are edited at the same time by two nodes, it will generate potential conflicts when trying to merge the changes. The author disabled the cache under such situation where each changes will be pushed immediately to the server. This behavior will incur performance degraded, but it ensures the consistency across nodes

5. I believe author has successfully shown that under their crafted environment, name and location transparency can be done and works well. However, it is on the assumption that network among machines are fast and reliable. It is reasonable to concern that when machines are on different locations, the operation delays will be so obvious that it is hard to hide from the users' perspective.

Xen

1. Xen is a high-performance hypervisor that allows multiple GuestOS to be run at the same time on a single machine. Xen is trying to make the GuestOS and the application inside believe they are running normally on a physical machine, which means the implementations should be hidden as much as possible.

2. There are several of them:

    1. Memory Virtualization: the GuestOS is still doing its own memory management. They can directly access the MMU, but all page fault and write operations needs to be validated and performed by Xen
    2. Compute Power: the GuestOS believes that they have direct control over the scheduling, however, Xen is managing the CPU time slice and using borrow virtual time to distribute CPU power
    3. I/O: All I/O operations are trapped and handled by Xen
    4. Ring: The GuestOS believe they are ring0, however, they are modified to be run under ring1, and Xen is in ring0. This behavior is true for X86 as it is support 4 rings.

3. Read file system: When an application tries to invoke a read() syscall, it is first trapped and handled by the GuestOS. However, as read() is a privileged instruction, Xen will trap it again when the GuestOS dispatches it, perform check to ensure it is from the right source as both application

and GuestOS is running at lower level, and emulate a virtual disk for GuestOS to continue action if pass.

4. There is a problem to fairly distribute compute power among GuestOS because even when an GuestOS has occupied CPU for and extended amount of time, we can't switch it off if it is on a critical section. In order to deal with it, borrowed virtual time(BVT) has introduced. GuestOS are allowed to borrow time from the future CPU allocation if they really need to perform some latency sensitive task. However, the more they requested, the heavier they are penalized in the future. E.g. having less chance to be executed.

5. I believe the Xen has done a pretty great job hiding its implementation over the application side. However, it requires modification over the GuestOS, and this would be an issue for administrators and users when each time an upgrade is required.

Grapevine

1. Grapevine is a distributed system with message and registration service. It tries to hide implementation of how the message flow between the services so that the client program only communicate with the Grapevine API and it appears locally
2. There are several part so them:
    1. User Package: Grapevine has a user package that allows client program to call and do various operation without know any of the details e.g. address of Grapevine server
    2. Messages: all clients are sending message to the nearest Grapevine server and forwarded by message services. Clients has no idea how the message go within the system
    3. Replica: Each user data will have at least two replica in the system to ensure reliability, while users have no idea where they are
3. Sending messages: when user A tries to send a message to user B using a client program based on Grapevine:
    1. The client program will send request using the Grapevine User Package, and it has no idea about any details about other Grapevine server
    2. The Library (User Package) will try to locate Grapevine server X, and send the message to its message service
    3. The server X will then locate the destination and forward the message to the optimized Grapevine server Y
    4. When User B retrieves the message, similar procedure happened: user package will locate the server and retrieve the message from the message services

Locating the nearest server can be done as all servers info can be retrieved by asking any of the registration service.

4. One of the issue will be the scaling issue. If all registration service has all information about the whole cluster, it will soon become the bottleneck of the whole system once more and more machine/services join the system, since it required each single machine to be able to handle all directory info. Author solved this problem by adding a level of indirection. Using steering list, the computation can be distributed into different servers as each of them will handle a portion of users. This limitation fixed the amount of compute power required, and expansion can be done by adding fixed power machines instead of required higher power machines, which will soon reach its limits.

5. I believe the author has give concrete samples/solution for various issues when doing distributed system, and hided them elegantly with Grapevine: Client only need to know is the Grapevine user package APIs. However, some of the actions are not well transparency. For example, changes are not immediately effective and need time to propagate. Some operations may cause heavy system load e.g. delete account will need all copies over the whole system to be deleted. Administrators are still required to understand how the operations works to prevent unwanted behavior/degraded.

(Lazy Evaluation) When implementing some service a common trade-off is between "eager evaluation" — performing an action immediately when it is requested (hoping to amortize that cost against future accesses) and "lazy evaluation" — deferring the requested action until some later point in time (assuming that it will cost less or may not need to be done in the future). What choice to make depends both on the cost of the operation and the expected workload.

1. For each of the following systems, identify an important operation that uses eager or lazy evaluation as an optimization.
2. Explain what the operation is and what the required semantics are for it to be correct.
3. Describe how the implementation is eager or lazy (including any mechanisms used to ensure correctness) and why this is expected to provide an optimization in the common case.
4. Finally, describe a workload scenario in which this implementation will not be an optimization.

3.1 Mach Shallow Object: this is a lazy evaluation (copy-on-write) example, a feature helps reduce unnecessary memory copy. Only modified pages are copies and changed by new spawn tasks.

For instance, there are two task shared one object in memory. The system will point to the same object for the tasks rather than copy one for each of them. If one of the task write to the object, instead of coping the whole object, a shallow object will be created. It will point all unchanged pages to the original object, and record the changed pages only.

It is a really clever approach and potentially saving lots of time and memory by reusing existing memory. However, shallow objects itself may be reference by another shallow objects, so this indirection may cause obvious overhead if the level of reference is too deep.

3.2 Exokernel Secure Binding: this is an eager evaluation example, a feature to reduce protection costs.

Library OSes are allowed to manage resources, and in order to isolate two untrusted library OSes, Exokernal does bindings and perform check on binding time. After that, access does not required additional check. For example, when doing packet filter, kernel will download code during binding phase, and the code will determine which package should be handled by library OSes during access phase. Without such mechanism, kernel will need to perform checks over every library OSes to look for its destination.

This approach reliefs the need of understanding the semantics of each access and let the OSes to handle them at their will in a secure way. However, overhead of constantly changing bindings may still incur performance issue. It may happened, for instance, multiple library OSes working on the same network interface. Kernel may need to re-download codes frequently for different communications.

3.3 Xen Asynchronous I/O rings: this is an lazy evaluation example, a feature to decouple GuestOS and Xen and allow better I/O arrangements.

When an domain requested an I/O operation, it places descriptor on the ring using the producer pointer, then Xen go ahead and handle them by removing them from the ring, and responses are put back in similar manner, where Xen is the producer and GuestOS is the consumer. The handling sequence is not guarantee FIFO, and each response with have an unique id to track. This allows Xen to have the freedom re-arrange I/O requests by scheduling policy or priority concerns.

This approach is pretty decent. The only flaw I could think of maybe that if a GuestOS has many request at once, it may overflow the ring, or taking too many space for Xen to keep the ring. Blocking mechanism may help prevent such abuse.

3.4 VAX/VMS Page Clustering: this is an eager/lazy evaluation example, a feature to avoid extra I/O caused by paging. Its avoids read/write small page size by batching operations, doing several page at once.

One example will be when the page fault occurs, pager will tend to load several pages into resident set at once rather than back and forth doing small pages. The range is determined by e.g. checking if the pages read is of an executable file over continuous virtual memory, then preloading it will help increase the execution speed. If data is read only until the page is requested, then frequent I/O operation will block CPU and lower users' experience.

Write operations, on the other hand, is a lazy evaluation since they are also delayed. VAX/VMS will only write page when it has to do so, and usually writes 100 pages on each I/O. Write delay can helps better arrange changes along with read operations, as well as avoid unnecessary write back if program terminates at the same time.

This approach may not work well on machine with limited memory space, which they may have to free up/write back all pages anyway for the current task.