Section 2: Week 5: Naming Schemes and Shared Memory

Nate Bachmeier

TIM-8120: Distributed Systems

September 22nd, 2019

North Central University

# Naming Schemes and Shared Memory

“There are three hard things in computer science, naming things and being off-by-one.” The classical joke applies to the design of distributed systems, as a mechanism for naming nodes within the system needs to exist. Modern systems use Domain Naming Services (DNS) to create a hierarchical structure to the enterprise; other scenarios can use protocols such as NetBIOS, for simple flat identifier lists. Similar to other distributed system concepts, the decision between a list or tree comes down to the required scale and complexity. Consider a small branch office with a dozen computers and a shared printer, there having the user-friendly name of *TedsLaptop* can efficiently rely on an *implicit contextual identification*. Then imagine a multinational corporation that has many Teds, and the need for an *explicit contextual identification* is required. Fully Qualified Domain Names (FQDN), e.g., *tedslaptop.seattle.wa.sales.contoso.com*, gives system designers the ability to express these contexts, and convey that “*Ted is a member of the sales team in Seattle, Washington.*”

# Naming Structure for Contoso

Contoso is a highly globally distributed organization, with resources dedicated to corporate affairs, manufacturing, and retail outlets (see Figure 1). They require a naming structure that allows explicit contextual identification of a resource within this environment. They chose a base naming convention, such as {city}.{country}.{department}.contoso.com. Within each branch office location, a need exists to pull similar resources into more fine-grained pools. For example, the Point of Sale devices in the Manhatten retail location might be named {device-id}.pos.nyc.us.retail.contoso.com.

Some of the services that are provided by Contoso are more global and can rely on implicit contextual naming. For instance, all employees need to authenticate against their local Directory Service endpoint. It would be confusing for these employees to remember this occurs on {node}.auth.{city}.{country}.ops.contoso.com versus the Cananocial Name (CNAME), auth.conotoso.com. As more traffic to services routes to CNAMEs, there is a virtualization of the addresses and an extension point to deliver more value to the caller. Consider a customer accessing multi-media content from the public website. If the Content Delivery Network (CDN) only streamed from New York, then users in Australia might complain about the latency. Instead, a Geo-Location Aware Edge can serve the requests, without coupling configuration into the website code. Amazon Web Services (AWS), a widely used public cloud platform, exposes these capabilities through their Route53 and CloudFront CDN offerings to efficiently and economically enable businesses of all sizes to leverage these scenarios.

# Memory Management Models

Three common strategies for allocating an address space are Single Continuous Allocation (SCA), Partitioned Allocation (PA), and Segmented Allocation (SA). The differences between these approaches can be made apparent through the example of allocating one gigabyte of memory to an application. Under an SCA solution, the operating system would create one virtual memory space that begins at some base address and ends 1E09 bytes later. The program can read or write to anywhere within this space and not concern itself with the physical memory pages below it. Those physical memory pages are an example of segmented allocations, each 4-32kb depending on the hardware architecture that needs to be managed by lower subsystems. These segmented allocations need to come from some pool (or partition) of resources, such as physical Random Access Memory (RAM) chips. When an operating system, like Windows or Linux, initializes, it will create these partitions for managing kernel space, disk swap space, and user mode stacks and heaps.

Many applications can rely on SCA functions, such as malloc; however, as the app becomes more interactive, using more advanced memory management operations is required for performance reasons. For example, a service listening process could use Shared Memory (SHM) for Inter-Process Communication (IPC), and then avoid redundantly copying network packet payloads as control transitions between process boundaries (González-Férez & Bilas, 2016). Instead, the network adapter creates writes the packet into SHM and then passes a pointer to the location. Other processes with access to the SHM object can then dereference the pointer if they need the contents. However, some specific applications are more performant through the use of message passing (Prvulovic, 2016). He argues that the ability to horizontally scale can enable the workload to span more resources and offset the overhead of additional serialization.

A critical challenge with SHM scenarios is managing the synchronization, or there is the potential for corrupt read scenarios. One solution is to use semaphores to limit which actor in the system is in control of a segment within the SHM (Kerrisk, 2015). Mendelsohn improved on this approach in Bloomberg’s Reader Writer Lock, which uses atomic accounting fields to track the active readers, pending writers, and a flag to indicate if the writer has control (Mendelsohn, 2017). A mutex uses these values to manage scheduling readers and semaphores to signal when a writer can start.

Another challenge comes from securing access to the SHM, as a malicious application could write corrupt structures with the goal of crashing other consumers. Windows-based applications can specify the scope of the SHM to be Global, Session, or User. Linux exposes a virtual file system that represents the state of the /dev/shm. Both Windows and Linux based systems support security permissions that limit which user accounts can open for reading or writing a handle to the underlying kernel objects (Gaztañaga, 2006). These protections can prevent malicious intent; however, they do not mitigate faults from inadequate protocols. Over the last decade, hundreds of critical vulnerabilities around the usage of SHM and Memory Mapped Files (MM), have been reported. An informal investigation shows these include target timing attacks, UI metadata, object name squatting, and incomplete argument validations too name a few (Lipp et al., 2018) (Horn, J et al., 2018) (Chen, Qian, & Mao, 2014).

# Using Shared Memory

Contoso needs to use shared memory patterns in their applications for scenarios such as efficient packet forwarding and distributing notifications to multiple consumers.

## High-Level Diagram

Figure 2 presents a high-level diagram of the workflow that starts with a client sending a request into a server’s network stack. The operating system places the payload into an SHM where an authorized user-mode process can read those details. The listening process can directly reference the data structure or mutate it using a copy-on-write strategy.

## Low-Level Diagram

Figure 3 presents a low-level diagram of the communication protocol between the writer and reader threads. These threads communicate across a partitioned SHM device, which contains one partition per computing unit. Writing to the SHM is divided into two logical steps: (1) inserting items into a FIFO queue and (2) dequeuing as access to the per partition locks (PPL) becomes available (Alappatt, 2018). A per partition reader thread (PPRT) then attempts to acquire the PPL and safely read the content. Next, the reader can invoke its implementation functions to carry out the operation.

## Expanding to Multi-Writer/Multi-Reader

Figure 4 presents an alternative model to support multiple readers on an individual partition. Similar to Figure 3, the writer horizontally scales across several SHM partitions to reduce contention. After checkpointing the content into the store, the subscribers are identified based on an externally managed Mapped File. Then a pointer to the shared memory is posted into a per reader per partition FIFO queue. A compute unit reader thread then pulls the next item and attempts to process it. As the queue depth increases, the number of PPRT can also grow to stay proportional and minimize latency (Sachs, Kounev, & Buchmann, 2013). However, increasing the number of concurrent PPRT can introduce challenges as the FIFO messages become out of order. To mitigate these scenarios, distributed consensus algorithms such as Paxos, can be leveraged to ensure a coherent single system view (Zhao, 2014).

## Improving the Reliability of Event Queues

Figure 5 presents considerations around fault tolerance of events as they traverse the system. Each computation unit can check out a message from the queue for a pre-defined duration. Using a time-based checkout model enables the system to recover if a stop-fault occurs in the message processor (Zhao, 2014). If the event can be processed correctly, then it is deleted from the incoming FIFO Queue. When a failure occurs, a decision needs to take place to re-attempt, terminate due to unknown failure reason, or archive due to too many errors. These transitional responses to error states need to archive into durable and economical storage for offline analysis. Depending on the availability and consistency requirements of the system (e.g., whether telemetry versus financial transactions), new processes and policies can be required.

# Figures

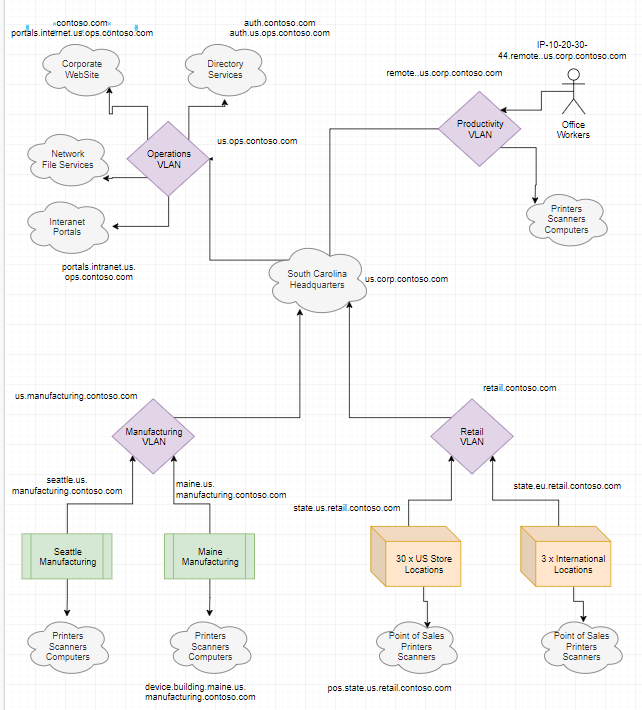


Figure 1: Naming Hierarchy

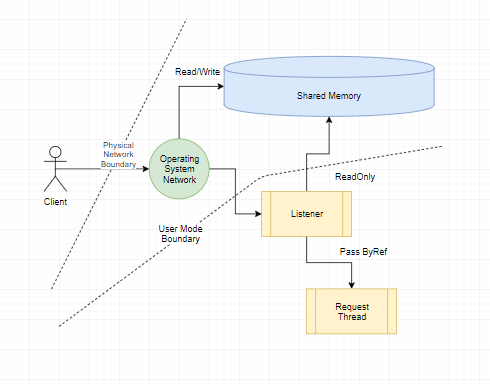


Figure 2: High Level Diagram

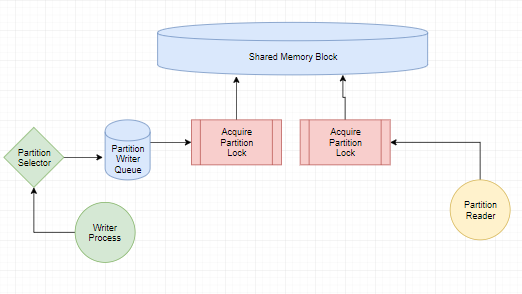


Figure 3: Low-Level Diagram

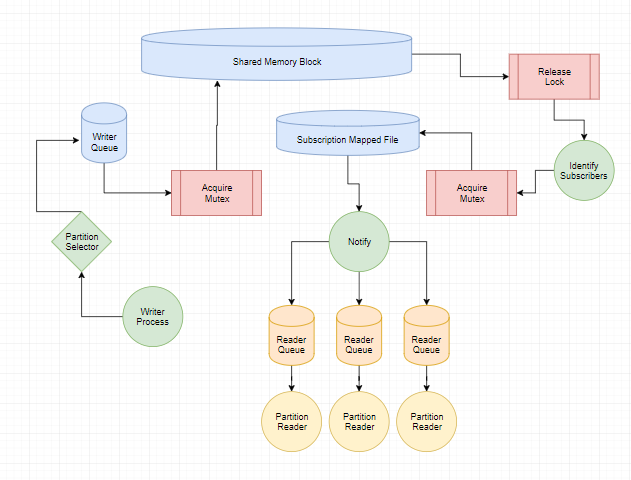


Figure 4: Broadcast Notification

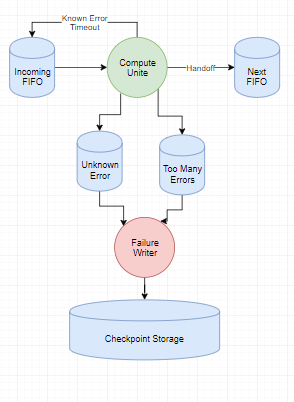


Figure 5: Fault-Tolerant Queuing

# References

Alappatt, A. (2018). Network Applications Are Interactive. Communications of the ACM, 46-53.

Chen, Q., Qian, Z., & Mao, M. (2014). Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks. Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, August 2014.

Gaztañaga, I. (2006, June 18). Memory Mapped Files And Shared Memory For C++. Retrieved from open-std.org: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2044.html

González-Férez & Bilas. (2016). Mitigation of NUMA and synchronization effects in high-speed network storage over raw Ethernet.

Horn, J et al. (2018). Meltdown: Reading Kernel Memory from User Space. 27th {USENIX} Security Symposium ({USENIX} Security 18.

Kerrisk, M. (2015, February). POSIX Shared Memory. Retrieved from Man 7: http://man7.org/training/download/posix\_shm\_slides.pdf

Lipp et al. (2018). Meltdown: Reading Kernel Memory from User Space.

Mendelsohn, J. (2017, October 23). CppCon 2017: Reader-Writer Lock versus Mutex - Understanding a Lost Bet. Retrieved from YouTube: https://www.youtube.com/watch?v=KJS3ikoiLso

Prvulovic, M. (2016). High Performance Computer Architecture by Georgia Tech. Retrieved from Udacity: https://www.udacity.com/course/high-performance-computer-architecture--ud007

Sachs, K., Kounev, S., & Buchmann, A. (2013). Performance modeling and analysis of message-oriented event-driven systems. Software & Systems Modeling. Oct2013, Vol. 12, Issue 4, 705-729.

Zhao, W. (2014). *Building Dependable Distributed Systems: Building Dependable Distributed* Systems. John Wiley & Sons, Incorporated.