# Section 1: Week 1: Distributed System Structure

Nate Bachmeier

TIM-8120: Distributed Systems

September 22nd, 2019

North Central University

# Part I: Distributed System Structure

Contoso Clothing is an international manufacturer and retailer of personal attire. Their manufacturing processes span three continents and need to meet the needs of their several thousand stores. Each location has dozens to hundreds of employees that need to have access to computing environments, point of sales services, and printers. There are multiple configuration options for the construction of this environment, each with different pros and cons. If they lack the understanding of these trade-offs, then the system will be (1) be too expensive to operate, (2) unable to meet peak loads and (3) unreliable during complex scenarios.

# Command and Control Structured

Traditional systems (see Figure 1) might use a static hierarchical structure where the headquarters will replicate policy through a tree structure (Steen & Tanenbaum, 2016). Within the tree there each branch represents an aggregation point, such as North America or Europe. There can be child branches to further distribute the load to management points, such as Washington and Spain. These management points will then execute commands on clients (leaf nodes) and collect any local results.

## Tree-Based Distribution Strategies

While these trees are good at distributing load across a vast breadth of systems, they introduce several single points of failure (Annadurai & Vijayalakshmi, 2016). To mitigate these risks, administrators implement these branch nodes as complex systems, not individual compute units (Khaneghah & Sharifi, 2013). For instance, the *Washington Management Point* might not be a single server but a load-balanced ring of servers.

By introducing a load-balancer, the administrators are trading availability for additional complexity. Consider the impact of a client sending three requests to the load-balancer, which in turn hands them to three different service instances. These scenarios can lead to (1) out-of-order eventing, (2) partial conversation failure, and (3) redundant resource allocations – to name a few challenges.

## Influence on Software Rejuvenation

However, it can simplify other scenarios such as software rejuvenation strategies as there are multiple identical processors within the functional group. Rejuvenation is the operational procedure of recycling private instance state after it has exceeded a threshold (Yang, Min, Yang, & Li, 2013). Since the functional group contains two or more nodes, the rejuvenation can be applied to one node while the other continues to service requests.

Perhaps the message processor leaks memory and becomes unresponsive after the working set exceeds 1GB. In this scenario having an external process (1) monitor the performance metric then (2) cleanly cycle the worker process as it approaches the threshold, would (3) increase the perceived reliability of the message processor.

# Distributed Office Systems

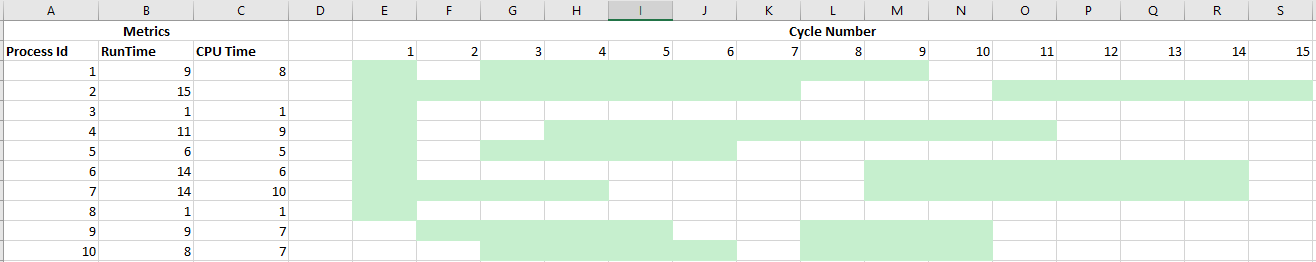
Enterprise environments also need to support business productivity centers (see figure 4) for their employees of the company. These topologies include devices such as printers, routers, point of sales systems, and laptops.

## Configured Heirarchy

Contoso might base this environment on the Command and Control structured, with the headquarters network administration team exposing core services, such as identity, name resolution, and virtual private network (VPN) gateway. Their headquarters could then securely connect to each branch office across the VPN tunnel to deliver network policies to the branch office machines. The branch office could then set a DMZ, such as a Reverse Proxy so that intranet and internet traffic remain separated. If the branch location offers Point of Sales devices, it would be advantageous to segment that portion of the network. This approach reduces the attack surface and helps to protect the resources.

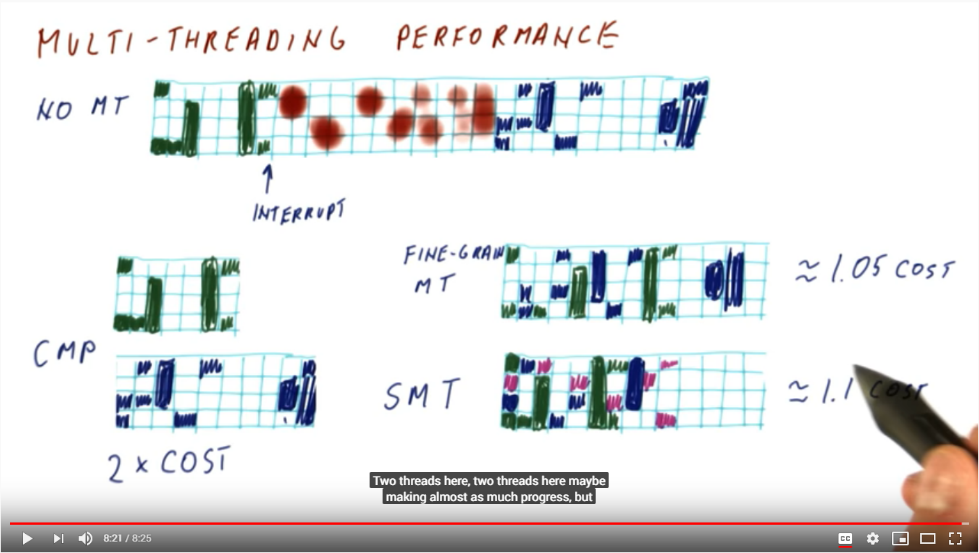
# Part 2: Processor Scheduling

Modern processors can run hundreds of processes in parallel, while only containing a handful of *physical* cores. Assume that ten very short lived processes are launched on an Intel Quad processor, with HyperThreading enabled. The operating system could then distribute load across eight *logical* cores. As these programs run there will be periods that nothing can occur, such as during a network call. The logical core will then be interrupted and attempt to make progress on another program’s unfinished work.



## Interweaving

Georgia Tech’s High-Performance Computer Architecture course includes this diagram to explain interweaving threads (Prvulovic, 2015).



At the top is the behavior of a single core and no hardware multi-threading support. For this scenario, a thread is scheduled and runs until an interrupt. After the interrupt, context switching takes place, and for some number of cycles, no progress made. Eventually, the second thread, shown in blue, can be scheduled.

To the bottom left is the behavior for a chip multiprocessor, which uses two physical cores to run two concurrent threads. A challenge with this approach is that it is prohibitively expensive due to requiring nearly twice the silicon.

The middle right is a fine-grained multi-thread core, that attempts to schedule something in every clock cycle. These processors do not wait for interrupts and evaluate instructions as soon as they become available.

The bottom right is a simultaneous multithreaded core (SMT) and is capable of running different threads during the same clock cycle. This scenario provides the most parallel execution as work is continuously flowing to all circuits as quickly as possible.

## Pipelining

Prvulovic provides an analogy that the instruction pipeline is similar to an oil pipeline. Let’s say that you pump a gallon of oil through the pipe, and that takes one day, how long does it take to get the second gallon? The answer is near real-time, provided they started at the same point. This behavior is because both gallons took one day, but the offset between them is near zero.

The same behavior exists for instruction pipelines as the processor attempts to feed in many operations from the different threads. For instance, the Hummingbird E203 will perform the services (stage 1) finger, decode, execute, write back; (stage 2) access via LSU; and (stage 3) copy back to a general register. (Tianchuan & Zhenbo, 2019).

## Speculative Execution

Modern processors follow the SMT strategy and use instruction pipelines as described above. Their goal is to minimize the amount of idle space within that pipeline as it is time that could be used to do something productive.

One strategy is to pre-emptively execute sections of the program that are likely to follow (Kocher, et al., 2018) (Lipp, et al., 2018). If the prediction is correct, then the results for that code block are already evaluated. Otherwise, the speculative execution is not committed, and the right instructions evaluated. Since the worst case of speculative execution equals the time of having been idle, there is no harm in doing it.

## Reorder Buffer

The densest and most complex component of the scheduling component is the reorder buffer (Choi, Park, & Jeong, 2013). This component is responsible for reassembling the out of order execution into the program defined order. Increasing the size of this structure can improve performance by increasing the amount of speculative execution; however, it also decreases power efficiency.

Choi et al. propose that one strategy is to move the reorder buffer into a separate component. Then exploit the fact that most basic blocks are around six instructions and by default correctly ordered. By optimizing for the typical case, and treating the exception as an exception, they can drastically reduce the power usage.

# Conclusion

There are multiple viable approaches to building distributed systems, and their pros and cons weighed.

Traditional methods have relied on command and structures such as tree-based distribution models. While these have been effective for decades, there are more modern solutions available through HPCS (and cloud) services. Orchestration Services segment the authoring problem into the business expert and system engineer domains. Through a clear separation of duties, the right experts can be more involved with the process and not hope the other teams did the right thing. These capabilities allow defects to be surfaced sooner and reduce the impact on customers.

Just as it is not possible to have a one-size fits all distributed system architecture, a distributed system architecture could use multiple patterns. Perhaps the Retail Services portal has a more natural alignment with Orchestration and the Manufacturing Services process better aligns with Data Processing Networks. In these scenarios, it should be perfectly acceptable to choose the right tool for the job. Ultimately the goal of a distributed system to resolve a program as quickly and efficiently as imaginable (Khaneghah & Sharifi, 2013). Provided that happens and the customer is delighted, not a lot else matters.

# Figures

## Figure 1: Command and Control

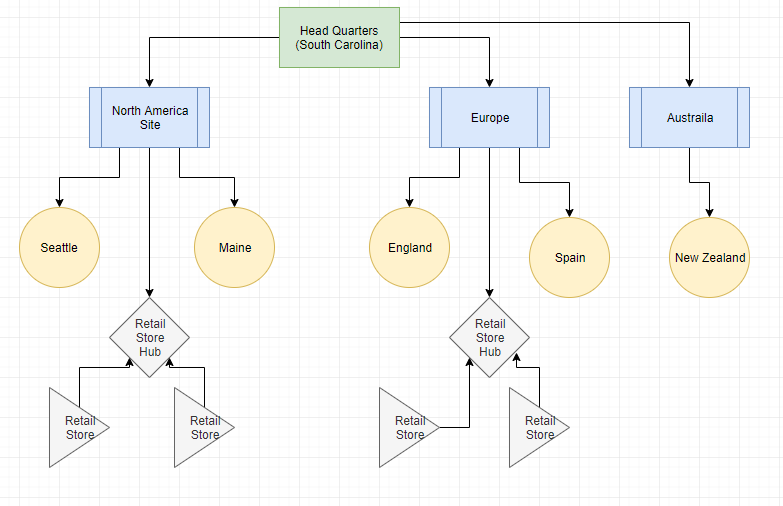
Traditional distributed systems might use a tree-based distribution model to propagate policy and aggregate results back to the top.

Figure 1: Command and Control

## Figure 2: Data Processing Networks

A command pattern for HPCS is to use publisher/subscription models to fan-out self-contained task descriptions. These descriptions are parallel processable, enabling enormous scalability.

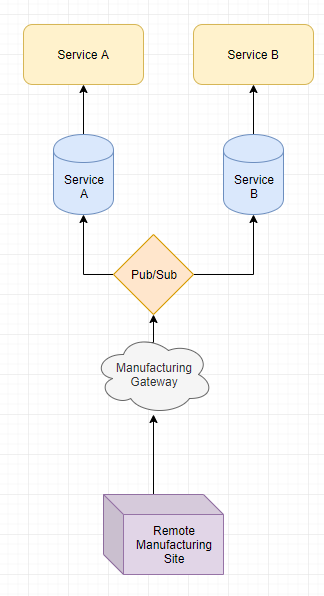


Figure 2: Pub/Sub

## Figure 3: Orchestration Services

This pattern leverages a central service to schedule remote calls to external services and apply compensation strategies for any failures.

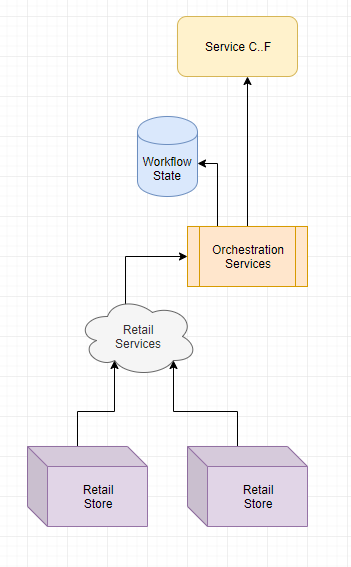


Figure 3: Orchestration Based

## Figure 4: Business Productivity Topology

Represents a typical corporate environment with branch office

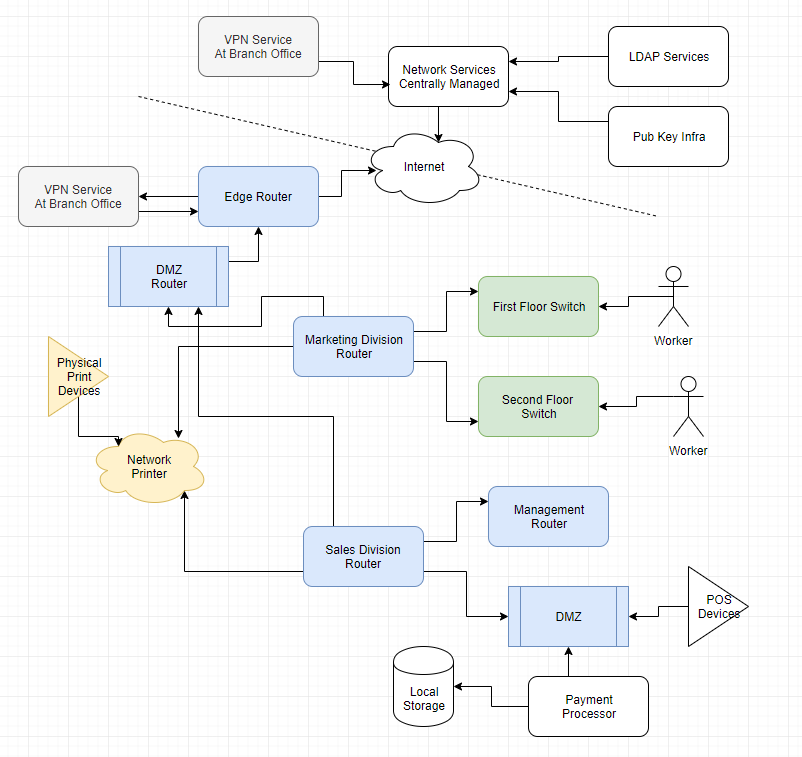


Figure 4 Business Productivity

# References

Annadurai & Vijayalakshmi. (2016). Context Centric Cluster Computing (C4) in Ad Hoc Network.

Baudisch & Schneider. (2013). Evaluation of Speculation in Out-of-Order Execution.

Celar, Mudnic, & Seremet. (2016). State of the Art Messaging for Distributed Computing Systems.

Choi, Park, & Jeong. (2013). Revisiting reorder buffer architecture for next generation high performance computing.

Keller. (2007). *BPEL in the Real World.* Retrieved from Oasis Open.

Khaneghah & Sharifi. (2013). AMRC: an algebraic model for reconfiguration of high performance cluster computing systems at runtime.

Kocher, Horn, Fogh, Genkin, Gruss, Haas, . . . Yarom. (2018). Spectre Attacks: Exploiting Speculative Execution.

Lipp, Schwarz, Gruss, Prescher, Hass, Fogh, . . . Hamburg. (2018). Meltdown: Reading Kernel Memory from User Space.

Prvulovic. (2015). *High-Performance Computer Architecture.* Retrieved from Udacity: https://classroom.udacity.com/courses/ud007/lessons/3650589023/concepts/9999288670923

Steen, V., & Tanenbaum. (2016). A brief introduction to distributed systems.

Tianchuan, & Zhenbo. (2019). An ultra-low power RISC-V processor pipeline structure.

Venkatesan, & Sridhar. (2015). A novel programming framework for architecting next generation enterprise scale information systems.

Yang, Min, Yang, & Li. (2013). Software rejuvenation in cluster computing systems with dependency between nodes.