COMPSCI 590S Systems for Data Science **Fall 2016**

Lecture 17: Latency-Tolerant Software Distributed Shared Memory

Lecturer: Emery Berger *Scribe(s): Jun Wang, Abhinav Tushar*

In this lecture we discussed Grappa, a distributed shared memory system and more generally, touched the differences between a shared memory (like Grappa) and a distributed memory system (like MapReduce).

17.1 Shared Memory vs. Distributed Memory Systems

A shared memory system works with a global addressing space for all the machines while a distributed memory system works with machine local addressing. Having different address spaces in distributed memory can result in issues due to concurrency.

Explicit message passing, instead of using memory read/write abstraction, can result in non deterministic message delivery. Moreover, concurrency due to multiple cores of a single machine is another source of uncertainty. These can result in data races and inconsistent world views for each of the machines. Debugging these distributed memory systems is also a problem. A debugger relies on a consistent global view to provide the following features:

- Breakpoints
- Watchers
- Stepping
- Memory inspection and manipulation

In a distributed system, there are multiple stacks, heaps and threads to keep track of and due to lack of a global view at any machine, it becomes hard to find and ascribe bugs.

17.2 Addressing in shared memory systems

Addressing in shared memory systems involves converting memory access instructions to network messages. For example if a program asks for the value of a variable x (say), the system has to figure out if the variable is mapped to current machine or is somewhere else. In first case, it will be a simple memory read. In second, it has to be a network message that fetches the data from a remote machine.

A global addressing space can be provided by either of these:

- Compiling the program in a way that considers the data partition among the machines and substitutes message passing for direct reads / writes whenever needed.
- Create a library which provides global classes which inherently are made to work with partitions.

In any case, there are performance issues due to locality. These are usually addressed by either:

- *Partitioning* the data among machines.
- *Prefetching*. A fetch-on-need scheme is costly as it exposes the network latency in the main program sequence. Aggregating small network messages, can hide the network latency under bandwidth.
- *Make it parallel*. Parallelizing the code is another way to hide latency. In a highly parallelized system, many pending network messages will not hurt that much because there will always be sufficient number of threads busy in actual computation, while the network latency involving other threads gets hidden.

17.3 Insights of the Grappa system

1. **New workloads**
   The workload of big data matches the good case of the shared memory. Rather than a single thread program randomly accesses memory, we have a lot of processor processing working over big data in embarrassingly parallel.
2. **New technologies**
   - *InfiniBand*
     InfiniBand is a an architecture and specification which features low latency high bandwidth It is widely used in high end servers.
   - *RDMA*
     RDMA, Remote Direct Memory Access, is a direct memory access from the memory of one computer into that of another without through either one's kernel. It provides high-throughput and low-latency.

17.4 Message Ordering

How to order events (e.g. messages) in distributed system? Events are naturally ordered in a single processor machine. But in a distributed system without global clock, local clocks may be unsynchronized and therefore events cannot be ordered by local times. However, our interest is not on obtaining and maintaining true time, but on getting event sequence that be agreed system-wide. To achieve this goal, "happened before" relation is developed: $a \to b$ means that a happens before b. If a is a message send and b is message being received, then $a \to b$ must be true. Lamport clock and vector clock are two algorithms using "happened before" relation.

17.4.1 Lamport clock

Lamport's algorithm:

- Each message carries a timestamp of the sending time according to sender's clock;
- When a message arrives and the receiver's clock is less than the timestamp on the received message, the systems clock is forwarded to the message's timestamp + 1. Otherwise, the message is recognized as out of order and nothing is done.

An example is shown in Figure 17.1.

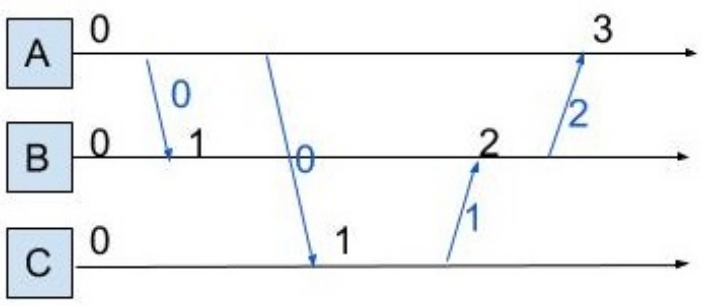

Figure 17.1: Example of Lamport clock

17.4.2 Vector clock

Vector clock is a generalization of Lamport clock. A vector clock in a system of N processes is a vector of N integers. Each process maintains its own vector clock to timestamp local events. Like Lamport timestamps, vector timestamps (the vector of N integers) are sent with each message.