A Cloud Powered Relaxed Heterogeneous Distributed Shared Memory System

Matías Teragni[1] , Gonzalo Zabala[1], Sebastian Blanco[1]

[1] CAETI - Facultad de Tecnología Informática, Universidad Abierta Interamericana, Buenos Aires, Argentina
{Matias.Teragni, Gonzalo.Zabala, Sebastian.Blanco}@uai.edu.ar

Abstract. Distributed systems allow the existence of impressive pieces of software, but usually impose strict restrictions on the implementation language and model. We propose a distribution system model that enables the incorporation of any hardware device connected to the internet as its nodes, and places no restriction on the execution engine, allowing the transparent incorporation of any existing codebase into a Distributed Shared Memory.

Keywords: Cloud Computing, Distributed Programming, Distributed Shared Memory, Multi-Platform.

1 Introduction

Modern applications have unprecedented requirements both in the available resources to them and the capacities associated with the platform that host them, including scalability and availability. The way we have transcended the limits of the current hardware is by building distributed systems.

1.1 Distributed System

A distributed system is a collection of independent computers that appears to its users as a single coherent system [1]. We understand a computer as either a memory module, a processor module or a combination of both. To build an application that makes use of all the available hardware, we can consider two main approaches:

Operative Level support [2], which today can be found as a cloud farm, where the distribution is hidden from the programmer, restricting the control over it

Application Level support, where the distribution is provided by a framework or an execution program, like a cluster built using Hadoop[1], forcing the developer to use specific languages and tools.

A Distributed Shared Memory [3] is a way of building a distributed system, by creating a shared memory system between the nodes and including an abstraction layer (either by software or hardware) that makes it work as a huge shared memory system, bringing the advantages in simplicity [4] and development cost of a single computer execution, and the scalability and performance gains of a distributed system [5].

[1] http://hadoop.apache.org/

To achieve this goal a Distributed Shared Memory includes a set of algorithms, protocols, and guidelines that allows two or more computers to work as if their combined memory cells were available for both to use. This enables computer programs to exist across more than one computer, and if built correctly, take advantage of having more memory and available processors.

1.2 Memory Model

A memory model is the axiomatic formalization of the legal behaviors [6] regarding memory access provided by an execution engine (either a physical computer or a virtual machine) and can be used to detect anomalies [7] and ensure the correct execution of a program according to the specified semantics of the language used to define it. Studying these issues becomes fundamental when the desired code can be executed in parallel [8], because most anomalies and errors that can be produced by a faulty model will not happen under the strict serialization imposed by having a single processing unit, but even in the case of a single processing unit, if more than a single process is executing concurrently, then some anomalies can arise as well. In practice, once an anomaly is detected and analyzed, the correct serialization techniques can be applied to the code, for example by ensuring that the compiler inserts fences or locks when they are required to guarantee the semantics of the program.

The cost of synchronization.

In the specific case of a multicore system, the serialization is an expensive operation. If one must stop every processor while the state of a register is replicated to each node, then every advantage that comes with having the multicore in the first place is lost [9]. In the case of a Distributed system, like the ones behind Cloud Computing, the cost must include the network latencies, making it even worse, and other failure condition that may arise, such as network partitioning, where a subset of the nodes get disconnected to the others, and can cause a divergence in the distributed system. These extra circumstances are the root of the CAP theorem, which establishes a limit in guaranteeing consistency, availability and partition-tolerance of a distributed storage system [10].

Relaxed memory models.

To address these issues and take advantage of the current available hardware, relaxed memory models are being used. A relaxed memory model [11] is one that provides certain useful guarantees of the semantics but is not as strict as a complete serialization of the operations, allowing different execution paths to be correct, and diminishing or eliminating completely the synchronization cost.

Eventual Consistency.

One of the most popular relaxed models is Eventual Consistency [12]. This model guarantees that every node of the system will, eventually, have the same state, even in the presence of network partitions or node disconnections.

This is incredibly useful for distributed databases (specially NoSQL ones) and content distribution networks, where the mutation of the data is limited, and the

robustness of the program is not compromised by discarded intermediate values. The only thing that matters in those applications is the last state.

Causal Consistency.

Causal consistency [13] comes into play as stronger restrictions are applied to the order of the operations, which must guarantee that some subset of them cannot be executed out of order. This can ensure the semantic of a given operation or algorithm but requires synchronization between the nodes. It is still weaker than a complete sequential order, because all the operations that are not related by a direct cause can be executed in any order.

To provide causal consistency the program code must be analyzed to correctly determine the happens-before relation that restricts the execution re-ordering. This can be achieved in a statically manner, analyzing the source code, or in a more dynamical one by having the code be interpreted, and that relationship determined at runtime.

Both methods can be built on top of the eventual consistency [14], but require the use of a special compiler, or a specifically designed execution engine.

Relaxed Distributed Memory.

Building a distributed system often requires standardized and controlled hardware and network arrangements but using a relaxed model could allow the creation of distributed systems over a wide area network, and heterogeneous devices, like the internet. The issue preventing the adoption of weak models for general purpose applications is that they may force the developer to be aware and explicitly manage the distribution and synchronization forbidding the use of several programming languages not equipped with the synchronization tools required to guarantee the correct behavior of the program. It leaks the abstraction provided by the distributed system.

In the following section we propose a distribution model to support this, including the specific requirements leading to it. In section 3 we discuss an implementation prototype, including the design choices made during development, after which we expose future work and conclusions.

2 Proposed Distribution Model

To address these issues we propose building, on top of the guarantees provided by causal consistency, a distributed system platform composed by a series of libraries or modules that can be imported into an existing codebase, enabling the transparent incorporation of heterogeneous nodes, over an internet connection, simplifying the development process by allowing the use of a general-purpose programming language. Allowing in the process the inclusion of hardware such as mobile phones, or IoT devices, and the use of different programming languages to define the behavior of the different components of the system.

2.1 Requirements of The Distribution Model

Transparent.

Transparency in the context of a distributed system is defined as hiding the fact that its processes and resources are physically distributed [1].When a distributed system is built on top of an unmanaged network, nodes can connect and disconnect at any time. Handling these contingencies adds complexity to the code. To avoid it, it is desirable that the nodes can connect and disconnect transparently, and the information exchange between them to be performed almost anonymously. This allows any part of the system to request actions without worrying about exactly which node will perform them and provides fault tolerance to the system.

Local Access Time.

The principal performance benefit to use Message Passing instead of a Shared memory resides on data locality [4] [15]. To take advantage of data locality and cope with the unknown network characteristics without giving an unpredictably slow access time, we must sacrifice memory on the devices by building a cache of the shared data, guaranteeing a low access time by ensuring that the data is already present on every node of the network, and making any access to the shared data work as a local access to the device memory.

Heterogeneous.

Considering that the network which connects the nodes of the proposed distributed system is the internet, the nodes themselves can be hardware of any kind. Today we have a huge number of smartphones and IoT devices connected that could be potentially part of a distributed system and including that hardware could provide a huge opportunity to capitalize existing processing power.

Partitionable.

If the system should be able to incorporate different kinds of hardware, each with its own resource limitations, then the distributed shared memory cannot be a complete snapshot of the program, causing some of the incorporated hardware, like a smartphone, to crash due to insufficient memory. Therefore the synchronized state should be a part of the whole memory, and each device should be able to copy just the parts that it needs.

2.2 Replication Scheme

Given the requirement that the nodes are connected over a wide area network, considering that the nodes of our distributed system can be the huge number of IoT devices and smartphones out there, building a peer to peer replication scheme over the internet would not be practical. First, the network devices that separate the local area networks from the wide area networks filter broadcast messages, making it difficult to build a real peer to peer communication system over a wide area network. In addition to that, if every phone would have to process incoming messages sent by every other phone out there, the exponentially big number of messages would saturate the network infrastructure and would enqueue faster than the devices could process them, creating a new source of latency.

2.3 Data Hub replication

To control the replication, a few nodes on the proposed network have the specific job of replicating the state of the application to every other node. When any node needs to modify the shared memory space it informs the corresponding data node, and the data node ensures the correct replication to every other node in the network.

The data node can serialize modification requests if they collide and can ensure the consistency of the data stored in the shared memory. To achieve consensus a data node will always have the last word over the specific data it handles.

These specific nodes could be deployed on Cloud Servers, making them responsible for receiving the modification requests and broadcasting those changes to the connected devices, building a distributed shared memory between the several hundreds of heterogeneous devices connected to them, and taking advantage of the power and availability of the current cloud infrastructure.

Requirements of the cloud server.

Current cloud infrastructure allows us to handle several hundreds of incoming connections, and provide the basis for a successful replication scheme, but to get the proper functionality out of a distributed system built this way, there are two guarantees that need to be provided on top of the basic cloud functionality.

Low Replication Time.

One of the most difficult problems to solve correctly is when a collision and anomalies occur, mainly because exactly how the system should resolve it depends heavily on the semantics of the programming language and specific situation.

A collision in this case might be defined as two nodes trying to operate on the same piece of information at the same time. An anomaly is a behavior that is not consistent with the sequential execution of the program. Since the information takes time to propagate over the network, then these problems are exacerbated because they cannot arise only when two nodes act at the same time, but when two nodes act inside a window of time smaller than the time required to propagate the information between them.

If the propagation of information is slow, more collisions will take place, and the system will lose coherency, breaking the abstraction of the shared memory. The system, then, must reduce the amount and weight of the messages that circulate over the network, for example using a technology like web sockets instead of HTTP request-response cuts the overhead of the messages sent by removing the unnecessary http headers on each piece of data that is moved around.

Optimized Replication Scheme.

The replication protocol can introduce latency by itself, every message sent by the nodes must be processed by the server, and every message sent by the server must be processed by the clients. The main risk in this case is that if any node on the network cannot process the incoming messages in time, creating an ever-increasing queue of messages to process.

This can be tackled using two approaches: we can reduce the number of messages that are sent, and we can reduce the processing cost of each individual message. If the data hub takes the responsibility of broadcasting the changes, then the clients do not need to incorporate a polling system to obtain the latest modifications, reducing greatly the number of messages that the server must process.

If the server consolidates the changes instead of sending each change as an individual message, then the number of messages the client must process is greatly reduced.

3 Implementation prototype

3.1 Datahub implementation

A prototype of a Distributed Shared Memory between heterogeneous devices was built on top of a Firebase Database (functioning as a Data Hub using our replication scheme). The client side was built on JavaScript because it can be executed on any device that can run a web browser, serving as a fair test ground of a distributed system on heterogeneous devices over a wide area network.

Cloud server options.

Most of the cloud engines available today (including those provided by Amazon[2], Microsoft[3] and Google[4]) provide different services, from hosting complete virtual machines to websites, services, or functional programs. They usually include also some sort of storage, either a relational SQL database, a NoSql document-oriented database, or a file system to store the information generated by our cloud application.

If we wanted to use one of these traditional providers as our data hubs we would have to implement a program that take advantage of the available storage to persist the information and we would need to implement the handling of the modifications, collisions, and replications. This means that we would have to implement the whole functionality required of the Data Hubs in the language and format supported by that specific host.

On the other hand, real-time cloud databases like Firebase[5], or Cloudant[6] provide an interesting service in which you have a NoSql persistence scheme that can be accessed simultaneously by several hundreds of devices, with the objective of building a server-less application, where your clients know how to handle those changes. This service is intended, initially, for mobile and web applications, but can in theory be used from any platform, assuming the implementation of the correct binding.

The NoSql data that is synchronized takes the form of a JSON defined tree, where the specific semantics of the nodes in that tree are defined by the user of the service,

[2] https://aws.amazon.com/
[3] https://azure.microsoft.com
[4] https://cloud.google.com/
[5] https://firebase.google.com/
[6] https://www.ibm.com/cloud/cloudant

this means that if we want to interact with this database we must define the structure of the data that is contained by it, to allow our clients to be aware of the changes received from this service. Additionally, it usually has an integration with the other Cloud platform services, allowing us to implement any functionality that we required on top of the data synchronization on the server side, like conflict resolution, or garbage collection.

Using this service, we can avoid implementing most of the requirements of our Data Hubs, since the persistence and synchronization of the data is solved out of the box, we only need clients that handle those changes correctly.

The prototype was built using Google Firebase because, unlike its competition, it guarantees that any modification performed by any of the devices is synchronized to every other device connected in a matter of milliseconds thanks to the connection between the clients and the database uses streaming http and web sockets to avoid unnecessary overhead.

Distributed Shared Memory using Firebase.

Although the service provided by Google handles the replication of the data among the devices, it does not constitute a distributed shared memory, to achieve this functionality an abstraction layer must be built and deployed on each client. This layer must process every incoming message with modifications to the synchronized data model and apply the equivalent changes to the local memory converting every change received from the Cloud Database into the corresponding mutation of the memory of the device. This conversion is closely related to the semantics of the client programming language, and the nature of the mutation to the data model.

The other responsibility of the abstraction layer is to detect every change of the local memory and impact an equivalent modification into the synchronized data model. Detecting the changes is not necessarily an easy task, since polling every memory section to be shared is inviable, the only way left to gain this functionality is to intercept the changes of the client programming language by using the tools available in that context. This issue has been addressed by creating proxies that allowed us to detect the modifications without a major performance penalty.

3.2 Heterogeneous Node Implementation

Defined Data structure.

An example was provided where each device handles the movement of a rectangle on the screen, but the complete state is shown to all the connected nodes[7]. The data structure on this prototype is composed by two sets JSON elements, an Array containing all the rectangles, and an Object for each rectangle, containing floats for the X and Y positions, and a string for the hexadecimal color value. Once connected a computer will create a new rectangle of a random color and add it to the rectangle set, after which a node can only modify its own rectangle but has access to the whole collection.

[7] http://hiveproject.github.io/Firebase/Demo/Square/

Prototype Functionality.

The implementation assumes that all the memory structures are local and have no explicit reference to the distribution, as seen in the code examples below. The only exception is the first line, that obtains a reference to an array present in the shared memory.

The first function inserts a newly created rectangle object (composed by x, y and color) into an array, that happens to be shared, and subscribes to the event related to mouse movement a function responsible of updating the position of that specific rectangle to the coordinates of the mouse in that device.

```
squares=hive.get("SquareDemoPosition");
…
let pos = {x:50, y:50, color:"#"+myColor};
pos= squares[squares.push(pos)-1];
canvas.onmousemove = function(e){
  var mouseX, mouseY;
  if(e.offsetX) {
    mouseX = e.offsetX;
    mouseY = e.offsetY;
  } else if(e.layerX) {
    mouseX = e.layerX;
    mouseY = e.layerY;
  }
  pos.x=mouseX; pos.y=mouseY;
}
```

Once a local object is referenced by a shared object, then the local object becomes a shared one, enabling synchronization of its data to every other device connected.

The second function handles Drawing, by clearing the screen, and going through the array of rectangles, painting them on an HTML5 canvas. This function executes periodically with a low interval to refresh the display.

```
var ctx=canvas.getContext("2d");
function draw(){
  ctx.fillStyle = "black";
  ctx.fillRect(0,0,500,500);
  for (let k in squares)
  {
    if(squares[k]) {
      let current = squares[k];
      ctx.fillStyle=current.color;
      ctx.fillRect(current.x-5,current.y-5,10,10);

    }
  }
}
setInterval(draw,10);
```

Guaranteeing the Program Semantics

Since all the shared memory can be accessed by any node in the distributed system, there is a gain to be made with the available parallelism, but as a drawback the risk of data races is high, and unpredictable or undefined behavior may occur. Usually to solve this a compiler or an interpreter adds fences or locks to guarantee the correct synchronization of the shared memory, but since this prototype was intended as a library, it cannot modify the execution engine.

Instead this implementation provides the tools required to do a mutual exclusion to the developer, and with them a compare and set and other synchronization techniques can be built. According to the demonstration in [16], it is not possible to have stronger consistency than causal consistency having a partitioned system without sacrificing availability or endangering divergence. Therefore, availability will be sacrificed to provide the desired semantics, and a critical section will never trigger unless the requesting node is currently connected to the corresponding Data Hub.

The lock mechanism, like the one provided by Java and other general-purpose languages, can be accessed as a function in the synchronization library, and receives two arguments.

```
hive.lock(object,callback);
```

- A shared object over which the lock will be held
- A callback function that will be executed once the lock is acquired

This function is asynchronous, because a blocking function would not work on the single-threaded JavaScript execution and guarantees that the callback will be executed only if the calling node is the owner of the lock. Once the execution of the callback is completed, all the modifications to the shared state performed will be pushed, and the lock is released automatically. If a node is disconnected, the callback will never be executed.

4 Future Work

The developed model and prototype serve as a proof of concept, and several issues still need to be addressed. For now, at least, the prototype can only synchronize the shared state, but it may be worthwhile to explore the synchronization of behavior, being able to send complete working objects from one node to the others. An optimization is required in the synchronization engine to avoid unnecessarily replication of the parts of the shared memory that are not accessed by the local node.

The locking mechanism still has issues regarding deadlocks, and a disconnection of a node that owns a lock does not release it, making it unavailable to everyone.

5 Conclusions

This work has proposed a model that enables building a distributed system platform that integrates heterogeneous nodes over an unmanaged network, permitting the incorporation of an existing codebase because the model works as a library, and without imposing a programing language, paradigm, or platform. The viability of this model was tested by the construction of a prototype that exhibits the desired features. To guarantee the correct execution, the correct implementation is left to the programmer,

and tools were provided to address the need of critical sections in a code. Looking forward, more experiences need to take place to correctly determine the application area where these kinds of systems can be more beneficial.

References

1. Tanenbaum, A., van Steen, M.: Distributed Systems: Principles and Paradigms. Prentice-Hall, Upper Saddle River, NJ, USA (2006)
2. Tanenbaum, A., Van Renesse, R.: Distributed operating systems. ACM Computing Surveys (CSUR) 17(4), 419-470 (1985)
3. Protic, J., Tomasevic, M., Milutinovic, V.: Distributed shared memory: concepts and systems. In : IEEE Parallel & Distributed Technology: Systems & Applications (1996)
4. Ngo, T., Snyder, L.: On the influence of programming models on shared memory computer performance. In : Proceedings Scalable High Performance Computing Conference SHPCC-92, Williamsburg, VA, USA, USA (1992)
5. Nitzberg, B., Lo, V.: Distributed shared memory: a survey of issues and algorithms. In : Computer - IEEE Computer Society (1991)
6. Manson, J., Pugh, W., Adve, S.: The Java memory model. In : roceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Long Beach, California, USA (2005)
7. Kwiatkowski, J., Pawlik, M., Konieczny, D.: Parallel Program Execution Anomalies. In : Proceedings of Large Scale Computations on Grids, Wisla, Poland (2006)
8. Boehm, H.-J., Adve, S.: Foundations of the C++ concurrency memory model. In : Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, Tucson, AZ, USA (2008)
9. Kuznetsov, P., Ravi, S.: On the Cost of Concurrency in Transactional Memory. In : Principles of Distributed Systems. OPODIS 2011, vol. 7109 (2011)
10. Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM SIGACT News 33(2), 51-59 (2002)
11. Boudol, G., Petri, G.: Relaxed memory models: an operational approach. In : Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Savannah, GA, USA (2009)
12. Burckhardt, S.: Principles of Eventual Consistency. Foundations and Trends in Programming Languages 1(1-2), 1-150 (2014)
13. Mosberger, D.: Memory consistency models. ACM SIGOPS Operating Systems Review 27(1), 18-26 (1993)
14. Lloyd, W., Freedman, M., Kaminsky, M., Andersen, D.: Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In : SOSP '11 Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, Cascais, Portugal (2011)
15. LeBlanc, T., Markatos, E.: Shared Memory vs. Message Passing in Shared-Memory Multiprocessors. In : Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing, Arlington, TX, USA, USA (1992)
16. Prince, M., Alvisi, L., Dahalin, M.: Consistency, Availability and Convergence. Technical Report, University of Texas, Austin (2011)