Entity based Programming Paradigm*

Nimit Singhania
University of Pennsylvania
nimits@seas.upenn.edu

ABSTRACT

Distributed systems are becoming increasingly important in order to scale services. However, these systems are complex with significant concurrency and need for synchronization. Hence, these are difficult to develop and we need better ways to program such systems. In this work, we present a new paradigm to program distributed systems based on real world distributed systems. We are social beings and are part of various teams and organizations or "real world" distributed systems. A fundamental approach we use to manage complexity of such real world systems is to abstract teams or organizations into single entities. We can use a similar approach in distributed systems to abstract a collection of nodes into an abstract *entity* and thus, make it easier to develop them. In this paper, we present this idea of entity abstraction and discuss various advantages of this approach.

1. INTRODUCTION

Programming is a process through which we design systems. A programming system creates an abstract world in which the programmer thinks and designs systems just like an artist's tools or a musical instrument. A good system is one that creates a world that is natural to think in and the constraints that the system imposes do not restrict the programmer's thinking. This enables the programmer to work in a world that is relatable and come up with simple and innovative solutions to problems. Thus, an important part of designing a programming system is designing a good abstraction for the programmer that makes it easy and natural to use. This is one reason why object oriented programming became popular over procedural programs. It provided a natural way to deal with the complexity of procedural programs by hiding the complexity inside objects.

Today, we are in the world of distributed systems where nodes collaborate with each other to exchange information and perform tasks. With uniprocessor speeds reaching their limits, such systems have become even more important to scale services. These systems are complex and difficult to develop and maintain. Further, careful tuning of various system parameters is necessary to achieve performance significantly better than uniprocessor systems [7]. Programming such systems is therefore difficult and no clean abstractions exist that can make this task easy. To tackle this problem, we present a new programming paradigm inspired by how "real life" distributed systems work.

We are social beings and we are in constant interaction with people around us. We are part of numerous real life distributed systems. Our friends and family are first such systems that we encounter in our lives. Then, as part of various teams, we work together to complete the task at hand. We are part of various organizations, where we collectively achieve goals with each person performing his/her small role in the organization. Hence, we have great experience working with real life systems and we can leverage this while designing distributed systems.

In fact, real life analogy is often used to describe distributed systems. When Leslie Lamport first described the Paxos algorithm for fault-tolerant distributed systems [5], he presented it as a protocol followed by a hypothetical Paxon parliament to run successfully even when many members were absent for a long period of time. Similarly, we often use real life analogues to describe components in distributed systems, for example leader/follower, live/dead, client/server etc, which makes it easy to visualize a system or protocol. Hence, real life systems play a critical role in our understanding of distributed systems.

Taking this analogy further, we define a programming paradigm where a distributed system is represented as an organization of entities. In real life, we organize ourselves into groups and organizations so as to make it easy to navigate and manage the complexity of the real world. Similarly, we can organize the nodes in a distributed system into teams and groups and thus create a hierarchy of entities that can make it easy to manage the complexity of the system. Just as the concept of object in object oriented programming made it easy to create complex software, the concept of *entity* can help create complex distributed software. We call this new approach *Entity based Programming Paradigm.*

*Comments and suggestions can be sent to nimits@seas.upenn.edu

The remaining paper is organized as follows. In Section 2, we elaborate further on the concept of entity. In Section 3, we describe a formal model for entities along with its various components. In Section 4, we describe how such a model can be compiled onto a network of nodes. In Section 5, we give some applications of this approach. In Section 6, we present some previous work on programming frameworks for distributed systems. Finally in Section 7, we conclude with possible future work.

2. THE ENTITY ABSTRACTION

An *entity* is an abstract unit that represents a group of nodes or sub-entities. It uses the services provided by its sub-entities and collaboration between these sub-entities to achieve its required goals. It has its own identity and appears as a single unit to the external world just as in real life a team or an organization is a whole unit and not just a collection of individuals. A distributed system is essentially a hierarchy of entities where each entity has a specific role and provides specific services.

The act of defining an entity helps us focus on the complete unit and not just the constituent sub-entities. This is necessary because, performance of an entity depends on the dynamics of the interaction between its constituents. Even though the individual components might perform well, if their actions are not coordinated and do not help the entity as a whole, the entity might perform poorly. Therefore, defining an entity helps us focus on improving the performance of the complete unit and not just the individual components. Further, having named entities helps us have compositionality and separation of concerns i.e. a programmer can organize the distributed system into a hierarchy of entities and focus on building one entity at a time from the constituent sub-entities.

The entity abstraction is very similar to an object in object oriented programming. The key difference between an object and an entity is that an entity is an active and a live unit while an object is passive. An entity consists of live sub-entities interacting with each other to provide a service and can possibly interact with the other entities. Whereas, an object consists of only static fields and properties that can be queried and manipulated by the external world. But, many insights from object oriented programming can be carried over to this programming paradigm. We can have classes and types of entities, where a class might provide specific services and functionality to the rest of the system. Also, we could define abstract entities which implement the core structure and some basic protocols for interaction between nodes and these could be extended further to realize the actual entities. Similarly, we could define interfaces that define a set of services. These interfaces could be implemented by multiple entities with different guarantees and based on the requirements, one of them could be chosen by the programmer to provide the required service.

Further, the idea of entities has already been used implicitly while designing distributed systems. The most relevant example is that of a web server. When we access a website, we rarely notice that there is a complete distributed system that works to provide us with required content. For example, a search engine consists of crawlers that crawl the web to gather information about the web, indexers that organize this information into categories and then query servers that based on our query, provide us with required results. Similarly, each website that provides us with content has a huge network of nodes working in the background. However, a website appears as an abstract single entity that provides a specific service. The real life analog of this is retail stores (say Macy's or Gap) that provide us with goods and merchandise and have numerous people working to provide us with this service. But, we rarely notice the people and look at a store as a single brand. Another example where entities have been in use is file system service (say HDFS) that is often used by distributed systems (for example computing systems like Spark) to store data. To the distributed system, the file system service appears as a single abstract entity almost like a hard disk in our computer even though it is a complex distributed system in itself. Hence, the entity abstraction has already been in use to manage complexity of software and is a relevant concept.

The act of encapsulating multiple units into a single abstract entity, be it a web-service, a retail store, a team or an organization gives us a high-level view of the system and helps us look at emergent patterns of behavior of the system. Thus, we believe it is a useful idea and can help us design better distributed systems

3. FORMAL MODEL

Now we present a formal model for an entity. This model gives a functional description of an entity and isolates the programmer from how it is physically implemented on a network of nodes. We address the compilation of entities on a network of nodes and related issues in the next section. The description of an entity can be split into its external and internal description.

The external description is the abstraction that an entity provides to the external world while internal description describes how this abstraction is implemented by the entity. As described earlier, an entity can provide certain services and can interact with other entities in the external world via certain protocols. A service might be a storage service, computation capability, a content delivery service etc. while a protocol might be a leader election protocol or a consensus protocol etc. It may also consume some services from the external world to perform its tasks. Hence, the external description consists of a set of services provided and consumed and a set of protocols that the entity may participate in. To the external world, it appears as if the entity is a single node that performs all these tasks.

As far as the internal description is concerned, we noted earlier that the entity uses a set of sub-entities to perform its tasks. Internally, an entity might consist of a set of variables that describe its current state and a set of sub-entities that perform various roles within the entity. Further, it might inform the sub-entities of the services available to them via other sub-entities and external world. Also, it might connect sub-entities that can interact and participate in protocols with each other. Further, the entity might assign specific sub-entities as listeners for service requests from the external world and communicators to participate in protocols in the external world. These listeners and communicators can

exchange messages with the external world and in-turn initiate local protocols.

We now describe each component in further detail. The external specification of an entity consists of the following:

- *Set of Services*: An entity might implement a set of services. A service is a task that the entity can perform when requested. It is like a remote procedure call to the entity and any authorized external entity can request this service. The request might be a message sent to the entity and when the request is completed, entity might respond back with the results (or not) based on the type of the service.

- *Set of Consumed Services*: The entity might also need services from the external world to implement its services and protocols.

- *Set of Protocols*: The entity might interact with its immediate neighbours via a set of protocols. A protocol is described by a set of entities with different or similar roles (say client/server, master/slave or peer-peer) and a set of messages that they can exchange. An entity might implement a protocol under a specific role and communicate with the other entities via such protocols.

The internal specification of the entity might depend on whether it is a single node or a collection of sub-entities. In the first case, the node itself might implement the services and the protocols that the entity provides. In the second case, the internal specification would consist of the following:

- *State*: The entity might consist of a set of common fields or variables that describe the global state of the entity. This may or may not be necessary. However, if necessary, we would have to think about where to place this state. It could be replicated between all sub-entities or stored at a designated leader sub-entity.

- *Sub-Entities*: The entity might consist of sub-entities of different types performing different roles. These sub-entities might be entities and thus create a hierarchy of entities.

- *Listeners*: The entity might register a set of sub-entities as listeners for each service provided by the entity. More than one listeners might be required in order to scale processing of service requests. In such a scenario, we might also need to specify the scheme for distributing requests among listeners. Further, the set of listeners might be changed dynamically.

- *Communicators*: The entity might similarly register communicators that communicate with the external world and participate in various protocols required by the entity. Again similar to listeners more than one communicators might be necessary and they could be assigned dynamically.

- Initialization: During initialization, the entity might inform each sub-entity of the services available to them via external world and other sub-entities. For example, if one of the sub-entities provides storage service, it might inform the remaining sub-entities that require a storage service. Further, it might connect sub-entities based on the protocols they participate in. For example, it might connect a master with every slave that it can communicate with via a master/slave protocol.

Apart from this, we can use inheritance (from object oriented programming) to define these entities in a modular way, i.e. we could define abstract entities that implement some basic protocols and services (for example leader election or consensus protocol) and then extend these to get the desired entities. Also, the above description only describes a static hierarchy of entities. It might be possible to create entities dynamically and add them to the hierarchy. Further, a distinction between control plane and data plane could be made i.e. some sub-entities might be responsible for managing the sub-entities within the entity while others might be solely responsible for servicing requests and messages. A large number of possibilities exist. However, we restrict ourselves to the simple model described above. In the next section we look at possible ways to implement this on a network of nodes.

Now, as an example, we describe the high level system design of Spark [9], a scalable computation engine in the above model. In Spark, there are two types of entities, a driver and worker. Driver receives requests from the user and sends computation tasks to the workers. It keeps track of the partitions of data stored in the local memory of each worker and uses this information to schedule tasks based on data-locality. If a node fails, then the driver schedules the tasks on other nodes. Workers in turn receive computation tasks from driver and either fetch data its local memory or from an HDFS file (external file service) and perform desired computation. In the existing implementation, Spark does not tolerate driver failures. However, in the above model, it is easy to replace the single node implementation of driver with a fault-tolerant implementation using multiple sub-entities. The fault-tolerant version only needs to ensure that the set of services and protocols needed are implemented correctly.

4. COMPILATION

In this section, we describe how an entity based design of distributed system can be realized on a network of nodes. Some of the associated issues are:

- Placement: An important aspect of compilation of the entity based design is placing entities on a network of nodes. Such a design presents a natural hierarchy where there is likely to be more communication between nodes within an entity than those across entities. Hence, nodes closer in the hierarchy of entities must be placed close to each other, while those far apart must be placed further. Further, nodes that participate in different protocols must be placed close to each other. While optimal placement of the nodes might be computationally hard, approximate techniques for example

Simulated Annealing [4] could be used to get close to optimal solutions.

- Addressing: Another closely related problem is that of addressing different nodes and entities in the system. The system design assumes that the nodes can be addressed directly via names or xpath in the hierarchy. However, once these nodes and entities are placed on a physical network, we need to map these xpaths to the physical address and this mapping needs to be stored within the nodes. Further, while entities representing single nodes might have a physical address, no single physical address may be present for entities that are collection of nodes. One possible solution is to use the addresses of its listeners and communicators to represent the address of the entity.
- Entity specific information: As described in the previous point, an entity consisting of multiple nodes may not be mapped to a single physical node. In such a case, it is not clear where to store the entity specific information like states and logs. One possible approach is to have a leader node for each entity that can manage the entity and store information specific to the entity. A drawback of this approach however is that the leader node can become a single point of failure for the entity. Alternately, we could have a backup leader node that can lead the entity if the leader node fails.

Once the nodes are placed and addresses are assigned to entities, the code for each node can be compiled using a standard compiler and we can have a working distributed system. Note that the above approach allows the programmer to design the system without thinking about the low level details of how the system is implemented on a network of nodes.

5. APPLICATIONS

In this section, we describe possible applications of this model. As described earlier, this model helps us abstract away individual components into entities and higher level entities and thus, gives us a way to focus on emergent behaviors of entities. This can be used in various forms for analyzing and debugging the system. Further, this model is quite synonymous with real world organizations and this can help programmers better visualize the system. Lastly, real world organizations have emerged after years of evolution and we can use ideas from these to implement better distributed systems.

5.1 Abstract View of System

An entity gives an abstract view of the underlying components. The behavior of an entity is a result of various events that happen within the entity and therefore represents the emergent patterns of interactions between components. For example, if the actions of individual components are synchronized and coordinated, the throughput achieved by the entity as a whole might be huge even though the individual components are slow. Whereas, if the actions of the components conflict with each other, they might lead to an overall poor performance. Similarly, even though the individual components might be faulty, the entity as a whole might appear to function correctly, if it has the capacity to handle faults. The otherwise is also possible, i.e. each individual component might function correctly, but if there is a bug in the protocol followed by the sub-entities, the entity might appear faulty. Hence, the dynamics of the entity can differ greatly from its sub-components. Note that, this is not true for objects in object oriented programming, because the behavior of objects depends deterministically on the underlying components and does not depend on dynamic events such as node failures and message drops.

This abstract view provided by an entity can be utilized in various forms. First, we can log events at each entity and thus, keep track of emergent behaviors. If a discrepancy is found, we can easily localize it to a specific entity such that its sub-entities function correctly. This could be used to debug faults and performance issues in the system. Also, if an entity is found to be adversarial, we could assign blame using the logged events. Further, we could use the abstract view to compare various protocols and algorithms and use it to improve the underlying protocols and algorithms. Apart from that, data collected at each entity and its sub-entities could be used to learn emergent patterns and fine tune the performance using machine learning techniques.

5.2 Real World Analogy

As mentioned earlier, there is significant overlap between how we view real world systems and distributed systems. The entity representation makes it even more concrete and makes it easy to visualize a distributed system as a real world system consisting of real world entities. This has many advantages. First, we can relate more easily with real world systems than the state machine based protocols. In fact, often the modes and messages of these protocols are named on real world actions, which makes them easy to understand. For example, in Two-Phase Commit, the modes Prepare and Commit represent states where the node is ready to commit and has made the commit respectively. Therefore, the entity abstraction would make it easier to design the distributed system.

Further, using this abstraction, we can transfer ideas from real world more easily. Real world systems have developed through years of evolution. For example, consider the trading system. In the beginning it was a barter system where trade involved actual exchange of goods. Then, the idea of money came in where goods were valued in gold or silver coins and later, symbolic money came into existence. There is a lot to learn from real world systems and we can implement the learnt ideas in distributed systems.

One such idea is timeliness. In the real world, in order to achieve predictable performance, organizations set deadlines for every task. The granularity at which these deadlines are applied might vary. For example, a big task might be broken down into smaller tasks and there might be a high-level deadline for the big task and low-level deadlines for each of the smaller tasks. Further, there might be penalties when the deadlines are not met. In the big picture, deadlines are crucial to synchronize the operations of an organization and have smooth and predictable performance. Also, failures to meet a deadline is often used by organizations to understand underlying problems and resolve them. We could apply a similar idea in distributed systems. We could set

deadlines for certain services provided by an entity and a penalty model in case the entity fails to meet the deadline. The entity in turn could set deadlines for its sub-entities and use these to meet its own deadlines. Further, it could reassign tasks to alternate sub-entities if one of the sub-entity fails to meet its deadlines. In the worst case, the entity might fail to meet the deadline altogether.

Many such ideas have potential to the transferred to distributed systems and the entity abstraction makes it easy to transfer them.

6. RELATED WORK

Significant research is being performed on designing and specifying distributed systems. One approach focuses on building general purpose distributed computing platforms that can be used by regular programmers to write parallel programs [9, 2]. Such systems operate on data parallel programs i.e. programs that apply similar operations on large amount of data and are useful for big data analytics. However, this represents only a small set of applications that can be achieved via distributed systems. Further, these systems are quite complicated themselves and could be developed modularly using the model described in this paper. Another approach focusses on using declarative languages to specify distributed systems [8, 6] Such languages might only be useful in domain specific settings and may not be expressive enough to specify a complete distributed system. Further, such languages may not allow fine grained control over the system and thus provide low performance. Then, some approaches use state-transition models to describe the system and protocols followed by the system [1, 3]. Such systems offer advantages of being easy to model check. However, as described earlier state transition models can be difficult to work with and may not provide a good abstraction to the users. Hence, the key feature that distinguishes our approach from existing work is the focus of providing a clean abstraction to the programmer. A clean abstraction is necessary for the programmer to come up with a good system design in the first place.

7. CONCLUSION

To conclude, we have presented a new approach to program distributed systems where the focus is on providing a clean and natural abstraction to the programmer based on real world distributed systems. We have presented the entity abstraction that enables a programmer to abstract a collection of nodes as a single entity and thus manage the complexity of the distributed system. This approach has many other advantages including having an abstract view of the system which could be used to debug faults and also having a real world interpretation that makes it easy to transfer ideas from the real world systems to distributed systems. In future, we plan to further refine the model and come up with a concrete programming language along with a compiler that can generate code for a physical network of machines. We also plan to use ideas from real world distributed systems to design better distributed systems.

8. REFERENCES

[1] G. Berry and G. Gonthier. The esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87 – 152, 1992.

[2] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2007 Eurosys Conference*, Lisbon, Portugal, March 2007. Association for Computing Machinery, Inc.

[3] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat. Mace: language support for building distributed systems. In *ACM SIGPLAN Notices*, volume 42, pages 179–188. ACM, 2007.

[4] S. Kirkpatrick, M. Vecchi, et al. Optimization by simmulated annealing. *science*, 220(4598):671–680, 1983.

[5] L. Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[6] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 75–90. ACM, 2005.

[7] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what cost? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, May 2015. USENIX Association.

[8] C. Monsanto, J. Reich, N. Foster, J. Rexford, D. Walker, et al. Composing software defined networks. In *NSDI*, pages 1–13, 2013.

[9] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, San Jose, CA, 2012. USENIX.