Actors for Distributed Dataflow

Benjamin Bengfort
University of Maryland
Department of Computer Science
Email: bengfort@cs.umd.edu

Allen Leis
University of Maryland
Department of Computer Science
Email: aleis@umd.edu

Konstantinos Xirogiannopoulos
University of Maryland
Department of Computer Science
Email: kostasx@cs.umd.edu

***Abstract*—Recently, the Actor model of concurrency in distributed systems has regained popularity as cluster computing frameworks for large scale analytics have started becoming mainstream. In particular, the use of virtual actors has shown to provide automatic scaling and load balancing through virtual actor properties of perpetual existence, automatic instantiation, and locale transparency. This programming model appears to be ideal for data processing of streams of unbounded data sets, in a computing architecture of live, online processing of data. In this paper, we present the communication patterns of three such data processing applications using sets (or "casts") of Actors. We then model the communication behavior on a cluster using a simulation and show that the virtual actor model effectively encapsulates how a cluster should behave in response to variable volumes of data. We propose that this simulation strongly motivates future work towards generalizing virtual actor spaces for distributed computation.**

I. Introduction

Sensors, web logs, and other timely data sources are increasingly being used for real-time or on-demand applications that utilize machine learning models to quickly make predictions and adapt to changes reflected by the input data. Timely sources present a challenge as they are streams of unbounded, occasionally unordered data sets that must be computed upon in an online, real-time fashion rather than in batch. When these data sources are also variable (e.g. the volume of traffic can occasionally spike) then the underlying computing framework must be able to adapt and balance the load or risk falling so far behind that the computation becomes meaningless. Scalability must be *effortless*; it should "just work" by simply scaling out the cluster size, and be *adaptive* so that the cluster can be fairly utilized by all available jobs.

Because of the widespread adoption of distributed computing frameworks like MapReduce [1] and Spark [2], distributed computing in a cluster of (often virtualized) economic servers has become the default, general method of high performance data processing. These frameworks primarily abstract the details of parallelism and distributing computation away from the developer, and have allowed a wider community of programmers and scientists to take advantage of parallel algorithms and libraries. Machine learning in particular has come of age in this computing environment, and as a result there exist many high level libraries for fitting a wide array of models in parallel.

As a result, tools for dealing with *streaming data* (unbounded, unordered, variable data sources) have mostly been contextualized similarly to parallel machine learning algorithms – via descriptions of how data flows through the application. Tools like Spark Streaming [3], Storm [4], and Google DataFlow [5] implement a *data flow* programming model, where analysis is described as a directed graph whose nodes represent a single computation, and whose edges represent the transmission of input and output values. Describing computation this way is simple, adaptable, and when mapped to a distributed topology, scalable. However, this model does not allow nodes to *communicate*, nor does it provide any flexibility in the computational topology for error handling, transactional guarantees, consistency, or persistence.

In response to this inflexibility of the data flow model, the *actor model* [6], [7] has been re-emerging as an abstraction that allows for more versatility in communication between distributed processes while still being at a high enough level to abstract the details of communication within the cluster away from the programmer. Ironically, it is this model that underpins the more general data flow models [8], perhaps due to the fact that these applications are implemented in the Scala programming language, which by default uses actors for concurrency [9], [10]. Actors in this context are high level primitives that do not share state, making them ideal candidates for under-the-hood parallelism. However, a simple data structure alone is not enough for distributed computing; there must also be an execution and job management context. Orleans [11] presents the idea of a "virtual actor space", analogous to virtual memory, such that actor activations (e.g. instances of a program) exist forever, are transparent to their locale, and are automatically instantiated and deactivated. Virtual actors therefore provide for automatic scaling and load balancing in the cluster, and we believe are a novel and suitable alternative to the data flow model in the context of online, streaming computation.

In this paper we investigate the potential of virtual actors for distributed computing on streaming data by simulating a computing cluster which implements a *generalized virtual actor space*. A generalized virtual actor space replaces the programing model of virtual actors with a daemon service that runs in the background of all nodes in the cluster; allocating resources to a variety of actor programs in an on-demand fashion such that the resources can scale as variability of data streams changes, while balancing load across many "always on" computations that must share resources. We have simulated this framework by analyzing the communication patterns of three real-time applications: a traditional data flow, an online recommendation application, and a solar weather

forecasting application. We then present an analysis of how the virtual actor model behaved in simulation on the cluster and show that this model is a suitable substitute for the data flow model.

The rest of this paper is organized as follows. In the first section we present the details of our distributed computing model using actors, and describe the generalized virtual actor space in detail. Following this, we describe the applications we analyzed, and how we derived a communication model from each type of application. Finally, we describe our simulation methodology, present our results, and conclude with a discussion and future work.

II. Distributed Computing Framework

This section describes a distributed computing framework that utilizes *virtual actors* and its implementation for handling always-on computations that deal with streaming data. This framework sets the stage for the analysis of applications and their communication patterns and informs the construction of the simulation.

A. Actors for Concurrency

Actors are a model for reasoning about concurrent computation which is inspired by the *communication mechanisms* of a society of experts that engage in effective problem solving [6]. Like other distributed models, actors are event oriented with "events" being messages received from other actors, and therefore parallelism is demonstrated through the partial ordering of these events [7]. At a higher level, actors can be seen as data or task parallel pieces of a larger overall computation which maintain their state (e.g. each actor is a SPMD program).

Actors embody the three fundamental properties of computation: (1) storage (state), (2) processing , and (3) communication. In the traditional model, actors provide a simple abstraction for units of computation that can be programmed. Using these three resources at their disposal, actors provide an intuitive way to reason about concurrent computation; as they can be easily associated with the real world. Actors provide a simple, concise interface which contributes to the programmability of this model.

Actors operate under three necessary and sufficient *axioms*. In response to the receipt of a message, apart from *altering its own state*, an actor can engage in one of three simple actions, they can:

1) Create a finite number of *new* actors.
2) Send a finite number of messages to other actors.
3) Designate its behavior upon receiving the next message.

According to the conceptual model, actors can process a single message at a time, but in practice, there are many cases where they can be implemented so that they can handle arbitrarily many messages concurrently. The power of this model is therefore its simplicity; the control flow of computation is concisely expressed and parallelism naturally "emerges" from the behavior of actor entities in response to the receipt of messages.

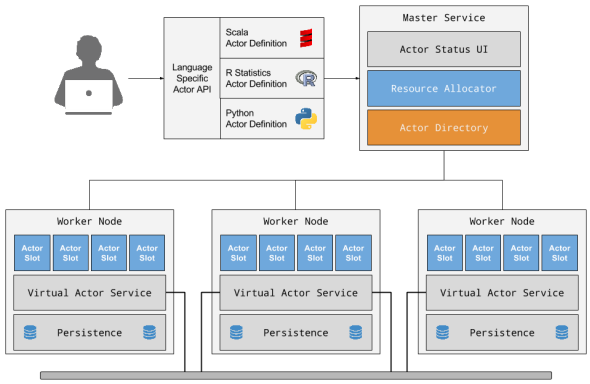


Fig. 1. The architecture for a generalized virtual actor framework.

B. Generalized Virtual Actor Space

The main source of inspiration for this work was the Orleans framework for building distributed applications [11]. Orleans' focus on *programmability* and *scalability* resonates with the modern requirement for demanding large-scale distributed applications that need to easily and continuously scale horizontally. The Orleans framework defines a model based on a modernized variation of the actor model, known as a *Virtual Actor Space*. In essence, a Virtual Actor Space consists of a global virtual address space, implemented as a distributed hash table that is used for load balancing actor instantiations, dealing with fault tolerance as well as maintaining high levels of availability across the board of usable nodes in the clusters.

The core ideas in Orleans that enhance its "programability" stem from the single premise that actors are logical entities that always exist for the duration of the computation. As a result, *actors can never be explicitly created or destroyed*. Actor entities have *activations* that get instantiated inside the system. Fault tolerance is provided by detecting message send failures – in this case the actor is reactivated on a different machine and computation continues normally. Finally, activations of actors can be either *stateless* or *stateful*. Stateless actors can be activated indefinitely upon as many nodes as needed for maximum concurrency, however, only one activation per stateful actor is allowed to provide consistency guarantees.

A virtual actor space therefore provides the perpetual existence of actors, automatic instantiation, and locale transparency. A generalized virtual actor framework can use these properties to automatically scale and load balance the activations running on a cluster at one time. However the true beauty of this method is that it does not require the programmer to consider the details of scaling and load balancing on the cluster. The user simply needs to define actors, their behavior, and how they communicate.

Recently, Orleans has also included specialized functionality for dealing with streams [12]; however this is simply an extension which holds true to the Orleans architecture geared towards distributed applications that incorporate streams of events. We believe that a *generalization* of this virtualization

layer can help sufficiently abstract distributed systems details, while using and extending the above ideas towards a *general* system for distributed computation. Our vision as shown in Figure 1 is a system where users define their computation as actors and actor behavior using the standard building blocks defined by the traditional actor model, while also including a Virtual Actor Space for abstracting away all distributed systems details and thus allowing for effortless and scalable analytics.

III. Applications Analysis

In order to show that a generalized virtual actor framework is a distributed programming abstraction well suited to handling streaming data for online applications we must show two things: (a) that the model is a high level abstraction, friendly to programmers while also providing powerful concurrent guarantees and (b) that the model can be applied to real world problems in an efficient and robust manner. We have begun to show the former in the last section where we described how programs are built using the actor framework, though additional future work such as an Actor based API would go farther to proving this. In this section, we explore the latter by analyzing how three streaming applications would be adapted to the actor model.

Our initial exploration led us to a wide variety of applications to simulate, from traditional HPC scientific computing applications like Lulesh to social applications like proximity notification services. As we begun to inspect these programs in detail, we noticed that many of them shared similar characteristics in their *communication patterns* – a fundamental aspect of actor based programming and streaming data in particular. In this section, we have identified three applications that typify the communications patterns we discovered. We have included a traditional data flow application for real-time email analysis, a rea-ltime recommendation analysis for an online store, and finally an application that uses an ensemble of machine learning to predict solar flare activity from magnetometer data.

In this section, we present a brief description of the application along with a definition of the *cast* of actors required for computation (e.g. an object level identification of the streaming processes in the application). We conclude the section with our identified communication patterns, as these patterns were primarily what we simulated to show that an actor based framework can be effectively applied to real applications.

A. Email Analysis Data Flow

The streaming data flow pattern was initially popularized by Twitter with the release of their open source Apache Storm project [4], which itself was followed by the Apache Heron project [13] after Twitter's real-time processing needs outpaced the low resource utilization of the Storm architecture. Both Storm and Heron allow the user to create a *topology* via a domain specific language (DSL). Topologies are directed, acyclic graphs of spouts and bolts, where a spout is a streaming data source that generates tuples of data to be fed into the topology, and bolts actually perform the computation upon those tuples. Bolts can transform incoming data and forward the information to other bolts downstream in the topology.

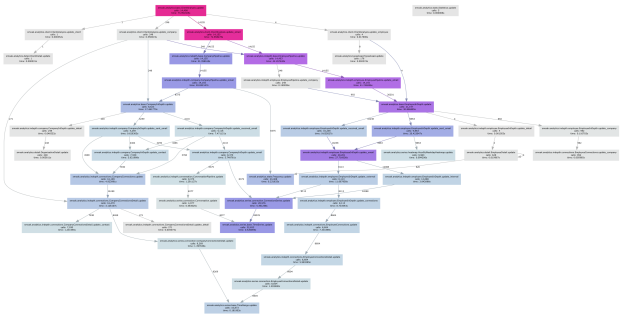


Fig. 2. The actor cast for a typical dataflow program.

Topologies in both Storm and Heron are essentially logical query plans that are then transformed into a physical query plan on a given system with a specific number of tasks. Shown in Figure 2 is the actor cast for a real implementation of a data flow for realtime email analysis. Topologies are often complex and conditionally executed based on the input data. Adding new tasks to a topology does not require modifying the entire data flow, but rather finding an *insertion point* where the input data matches the specification of the new tasks, and creating a new data flow. As such, topologies themselves are often dynamic, continuously responding to changes in the requirements of an application.

Because data flows provide data parallelism, concurrency is available in two places: different processes can process the same tuple in different points of the topology simultaneously and multiple processes can implement a single bolt. Scalability in a data flow system involves the management of executors and communication in a topology, which is the fundamental difference between the Heron and Storm architectures. A primary concern in this type of architecture is that the *message passing* cost of processes distributed across the cluster can quickly overwhelm the benefit of parallelism, as processes do more waiting on messages than actual computation. However, when implemented correctly, this model is currently the defacto method of handling large scale streaming data, and thus a virtual actor framework should be able to implement similar computations just as well if not better than the data flow standard.

B. Online Recommendations

The primary motivation for the use of the actor abstraction for distributed computing is to provide a programming model that allowed cyclic or direct inter-process communication between stateful nodes, something that the data flow model as proposed in the previous section cannot allow. This requirement is primarily related to online machine learning methods that utilize some form of optimization to fit a model, particularly linear models, clustering approaches, or training deep neural networks. Therefore, the second application we analyzed was that of a collaborative filtering system that

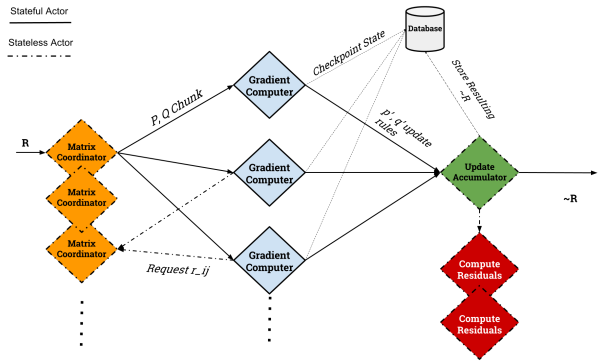


Fig. 3. The actor cast for online recommendations with matrix factorization.

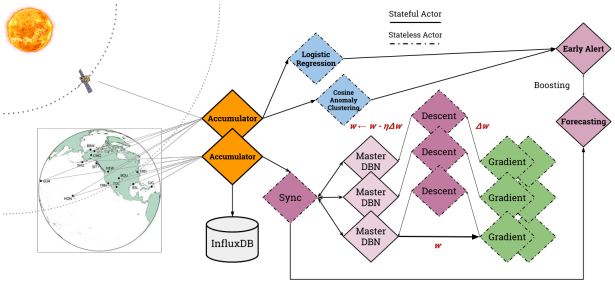


Fig. 4. The actor cast for boosting multiple models for solar flare prediction.

provides real time recommendations for users as they browse an online store, viewing items or adding them to their cart.

The most popular form of collaborative filtering for recommendations that is currently implemented are variations of the Netflix prize, which used a non-negative matrix factorization method with temporal dynamics [14]. Collaborative filtering involves the use of very sparse matrices, and sparse matrix factorization techniques for high performance computing is well studied [15]. Currently the most popular algorithms used for parallel matrix factorization in recommenders are alternating least squares (ALS), which is implemented in Spark MLlib as well as cyclic coordinated descent (CCD) [16].

Although both ALS and CCD could be implemented as actor casts, we have chosen to present an actor cast for distributed stochastic gradient descent (SGD) [17] as shown in Figure 3. Distributed SGD is more generally applied and well understood. In this model, accumulators are required to buffer incoming streaming data while the model is being computed upon and multiple models can be computed simultaneously depending on data volume and partitioning schemes (for example by genre for a book store).

A set of actors decomposes the original, sparse matrix and begins the process of updating the divisors of the original matrix based on a learning rate and the derivative of the error curve. However, each gradient computer must also send horizontal information based on their computations to ensure that the dot product across the entire matrix is computed correctly, requiring interprocess communication. An accumulator aggregates the matrix tuples, which can then be fed to a parallel residuals computation. Although the details of this model are omitted for brevity, we want to primarily show that parallel and distributed algorithms can easily be implemented using an actor model, and in this case, using a cast of relatively few types of actors.

C. Solar Flare Prediction

The final application we analyzed is the most complex, and presents a significant challenge to both the traditional data flow model, as well as online machine learning. The inspiration for this analysis was a novel streaming data source, namely the 14 magnetic observatories across the northern hemisphere that monitor the earth's magnetic field for the USGS Geomagnetism Program [18]. Traditionally, the earth's magnetic field is modeled using scientific computing methods, however we believe that real-time Bayesian anomaly detection techniques [19] might allow us to *predict* incoming solar weather based on currently observed anomalies.

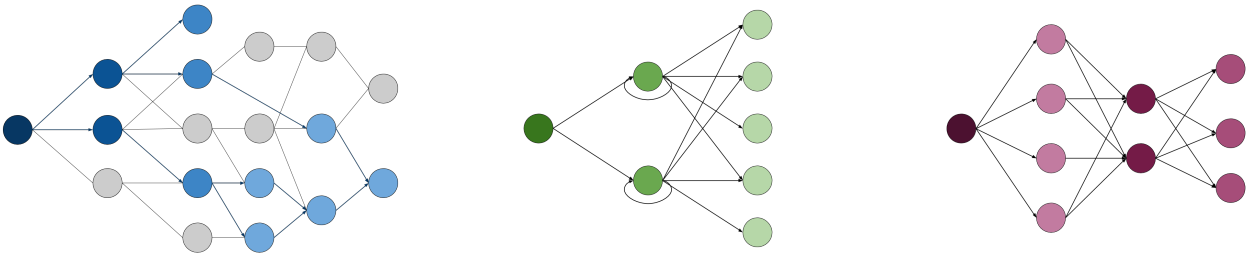
Bayesian approaches to solar flare prediction have been proposed in [20], which notes that past activity of flares in a particular region of the Sun is a good indicator of future occurrences. Correlation of sunspot groups and flares using commonplace machine learning techniques was proposed in [21]. Furthermore, an analysis of short term sequential magnetosphere data (of the sun) was analyzed as features to supervised methods in [22], showing that three days worth of solar magnetosphere data was enough to predict flares; this approach was made more accurate using a multi-resolution (e.g. boosting) approach that required less historical data to predict flares [23]. None of these studies showed the relationship of predictors based on measuring the Earth's magnetosphere, however, we believe they are sufficient motivation for investigation into an actor cast implementation.

Although these observatories provide data in the form of vectors of the earth's magnetic field at a near constant rate, we propose that the on-demand scaling of a generalized virtual actor space is still useful in the context of an *ensemble* of online machine learning methods. Here weaker, more compute efficient methods are used to detect anomalies at a broad granularity, while data is fed at a more stately rate to stronger, more compute-heavy mechanisms. However, when the weaker methods detect a potential anomaly at the higher granularity, the stronger methods are scaled to get a more precise outcome.

The actor cast shown in Figure 4 utilizes two weaker models (here logistic regression of time series information and cosine anomaly detection) to scale a Bayesian Deep Belief Network (DBN) on demand. The DBN is trained using yet another implementation of distributed optimization, here Downpour Stochastic Gradient Descent [24], which is specifically designed for large scale deep neural and deep belief networks.

D. Communication Pattern Analysis

As we analyzed the above applications we began to notice that they broadly defined a more general communication pattern. Actors are inherently about communication, therefore our approach was to first model the application as an actor cast,



(a) Email Dataflow (b) Online Recommendation (c) Solar Prediction

Fig. 5. The corresponding communications patterns that we developed from these applications included (a) branching, partial communication; (b) iterative, front-loaded communication; and (c) multiple-channel, full communication. Our simulation attempted to replicate these patterns with Actors.

then abstract each cast into a more general communication pattern. We then used these more general communication patterns as a guide to simulate the implementation of these applications. By simulating the general communication pattern, we hope to show that we can cover a wider array of applications and show how generalized virtual actor space load balances and scales in the face of variable data streams.

Figure 5 shows the corresponding communications patterns we developed from each application. The email dataflow application (a) shows a communication pattern that is unidirectional and that includes branching, partial communication. In this application a message is replicated across the cluster, usually less times than there are actors. The online recommender (b) however, shows a communication model that is front loaded, and iterative; here more messages are generated during processing than those that are incoming. These messages are then passed downstream to finalizing actors where there is more balance. Finally, the solar prediction application (c) instead reflects a multi-channel, full communication approach; where data is buffered at the front (e.g. the weaker models) then boosted through the stronger models (representing a bottleneck) before being output at the end.

While these simplifications necessarily gloss over the computational requirements of an actor cast implementation of these applications, we believe that a focus on the communication patterns of actors represents the weakest point in a generalized virtual actor space. Techniques like speculative execution and heartbeat monitoring of processes allow for fault tolerance inside of communication, however the primary scaling and load balancing techniques are based solely on *message load* and *throughput*. In the next section we will discuss how we implemented our simulation of actor communication.

IV. Simulation Methodology

Our primary method of investigation and analysis was a *discrete event simulation* of network communication in a cluster between actor programs. Our implementation was written in Python using the `simpy` [25] simulation library. In brief, `simpy` uses an event model that leverages `generator` objects (iterables or functions that `yield`, relinquishing processing until their `next` method is called) in order to control the state of the simulation. Discrete event simulations are therefore able to sequentially model complex systems of events that occur simultaneously, making them a useful tool for modeling network communication behavior.

Our primary framework was composed of two basic object types: `Process` and `Resource`. `Process` objects wrapped the `simpy.Environment` and implemented a `run` function that could yield events throughout the course of the simulation. Our `ActorMananager` and `ActorProgram` objects as well as our `Stream` dynamo were the primary process objects implemented in the simulation. The `Resource` objects wrapped a `simpy.Container` and represented a finite set of resources available to various processes. The most notable resource in our simulation was the `Network` object which provided limited bandwidth and increasing latency for message passing.

The overall framework simulated a `Cluster`, which was composed of `Rack` objects that contained a number of `Node` objects as well as a `Network`. The `Network` provided a realistic simulation of message delay due to latency as well as congestion. Communication between nodes in two different racks was penalized (the "intra-rack communications penalty") while communications between nodes on the same rack was much faster. Both racks and nodes implemented an interface for message passing to ensure that communication was handled correctly. Nodes contain other resources as well, including a fixed number of CPUs (for the purposes of this simulation, we modeled one process to one CPU) as well as a fixed amount of `Memory`. Although we didn't go deep into modeling computational behavior, we have left stubs to simulate more traditional HPC implementations in the future.

A. Data Generation

An essential piece of our simulation is the generation of variable, streaming data into the cluster. We required a "spikey" model of data volume, one where normal traffic is routine, but occasional spikes of activity overload the system. Data was generated using `Dynamo` objects, which randomly generated values on request. Our two basic dynamos included a uniform distribution as well as a Gaussian distribution of data generation.

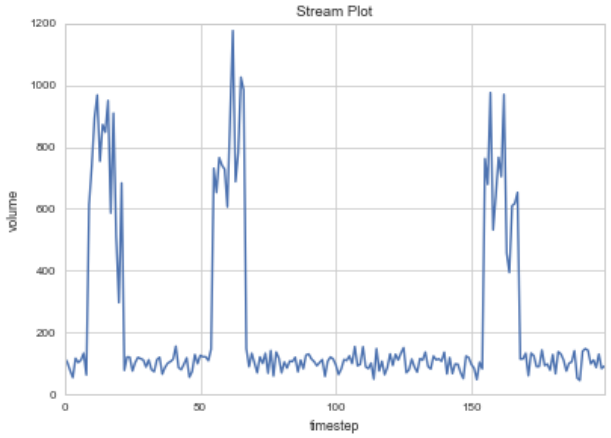


Fig. 6. Variable streaming data source with spikes of high activity (volume) that was scaled from longer periods of normal activity.

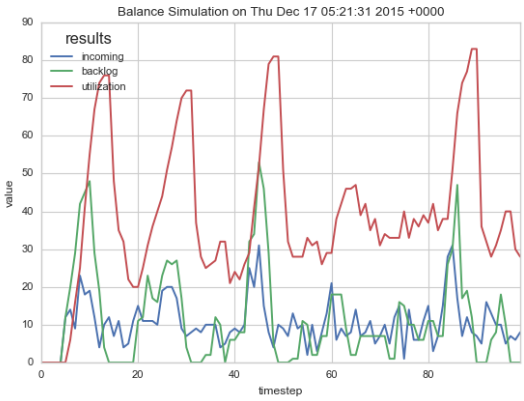


Fig. 7. Actor service load balancing the number of active actors based on a variable rate of messages. This figure shows the utilization in terms of number of actors in response to the number of incoming messages (spikes) as well as the backlog of queued messages.

Our streaming data dynamo was a bit more complex, however. It's basic operation was to create a normally distributed message volume, and took as parameters a mean number of messages and a standard deviation. At each step in the stream, with some small probability a "spike" could occur. The spike scaled the normal data volume by a uniform random amount, for a uniform random amount of time. The effect of this is a variable, streaming data pattern as shown in Figure 6.

B. Actor Management

Virtual actors were implemented in our cluster by means of a centralized `ActorManager` service, to which all messages were routed. Upon routing, the actor manager checked if the message was specifically addressed, and if so, ensured that the associated actor was activated and ready to receive the message. If not, the actor manager used a heuristic function to look for the "best" available actor to forward the message to ("best" being a function of distance, availability, and readiness). Given no active actors, but available space in the cluster, the actor manager activated dehydrated `ActorProgram` processes in preparation for forwarding incoming messages to.

The actor manager also served a vital *load balancing* role as shown in Figure 7. The actor service maintains a "backlog" or queue of messages that haven't been routed yet, either due to availability or delay in instantiation. If the backlog is empty, the actor service compares the number of available actors to the number required for handling messages. If there are too many available actors, it sends deactivate commands, which persist the actor's state (a fixed time cost) then shut down the actor. Figure 7 shows that the utilization (number of actors) is offset, but proportional to spikes in the streaming data service.

The behavior of each `ActorProgram` was to maintain state - either as activated/deactivated and as ready/busy. In the ready state, the program could accept messages, and "handle them" by imposing a variable delay cost (processing time) and then by sending zero or more additional messages, routing them through the actor service. Communication patterns were implemented by identifying different actor programs as *colors*. Actors of different colors sent a variable number of messages to actors of other colors based on their rank in the communications hierarchy.

For example in the online recommendations communication pattern, the `green` actor (the root node of the graph) sent two messages to `forest` actors. Each `forest` actor sent a message to another `forest` actor with a decrementing count. Once the counter reached zero, the message was forwarded to several `lime` actors. By using color encoding, multiple simulations of applications could be run simultaneously. For example, by having a streaming data service source both `red` and `blue` messages, then both communications patterns would exist on the cluster simultaneously.

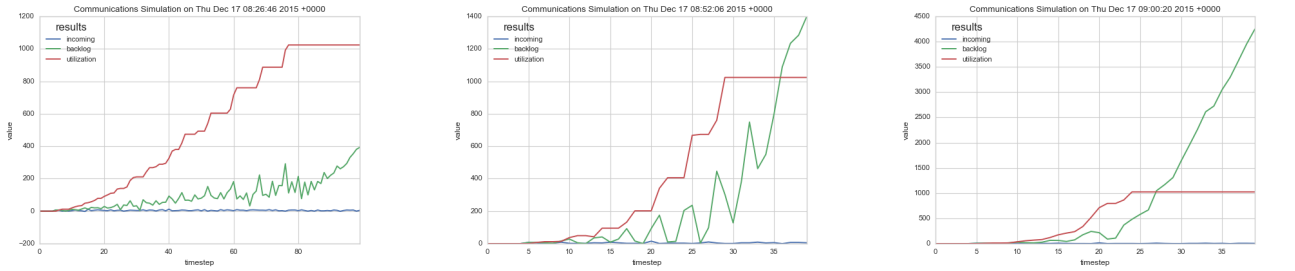
V. Discussion

Utilizing a simulation to study the computational framework we devised allowed us to isolate the communications model of the actor framework. However, the complexity of performing a discrete event simulation combined with specific requirements for each communication pattern led to challenges in getting our implementation set up and running correctly. We were plagued by a variety of bugs that were extremely difficult to track down as the simulation buried exceptions as "failed events". As a result, even though we did manage to simulate a variety of communication patterns, we ran out of time to start inspecting simulation configuration variables and perform a more thorough analysis on our results.

However, along the way we did learn a lot about how to implement both an actor simulation as well as a prototype virtual actor framework. In this section we will present several scenarios that we encountered during our simulation that required framework-specific solutions. We will also present our goals for future work.

A. Results

Unfortunately as shown in Figure 8 we can only display initial results for the three communication patterns we have



(a) Partial Communication (b) Iterative Communication (c) Multi-Channel Communication

Fig. 8. The initial results for utilization and queueing relative to message load for our three general communication patterns.

implemented in simulation. This is partially due to hard to track bugs in the system, but also due to the unique load balancing and scaling challenges involved in each communication model. The primary lesson learned from our study is that although actor based communication provides a simple model of concurrency (and even a simple method for simulation), *virtual actors* require more engineering related to backpressure of incoming data, and the algorithms used for both scaling and load balancing.

In many cases, a configuration search is required to find the optimal number of nodes to handle both normal traffic and spikes. As shown in Figure 8c, if the number of available actors doesn't exceed the expected traffic, then the queue will continue to grow until the entire capacity of the cluster is consumed. Even if there is buffer, that is times when the traffic is light enough to allow the actors to begin to recover, if the cluster doesn't scale in-time, it will get far enough behind that it cannot recover.

As expected, in each communication pattern the number of messages generated by actors far exceeded the incoming messages, as data was passed through the communication flow. As in Figure 8a, if the ratio of the maximum number of nodes relative to the messages generated is proportional to the incoming messages, then the simulation plateaus, are only just able to keep up with normal message traffic. However, in the iterative case as in Figure 8b, even when iteration is complete (as evidenced by the spikes in the queue), the lag behind the system state and activations causes more actor activations, never allowing the system to balance properly, the backlog starts to grow out of proportion.

Both of the cases described above are due to the mismatch between the *activation time* and the *message receipt*. We tried various solutions, including introducing a message lag (which had the effect of never allowing the queue to drop to zero), as well as tracking the number of activations and only allowing a proportion to be activated (which didn't limit the peak activations, but made the system take longer to get there). In either case, it is clear that more sophisticated protocols than the ones we implemented are required for a prototype virtual actor framework.

B. Future Work

The semester long task that we undertook to study the use of actors for streaming data was certainly an ambitious one, which had many twists probably reflected in this paper. We didn't conclude to do the simulation until much later in our study, focusing instead on attempting to implement a parallel implementation of one of the applications we have analyzed. As a result, our simulation results are not as robust as we would like. Although we did manage to simulate general communication patterns, further study in simulation is required to show how variable load parameters affect utilization.

We therefore have three primary goals for future work: continuing the simulation, developing a virtual actor API, and building a generalized virtual actor system. By continuing to add complexity to the simulation we hope to encourage more precise results that better demonstrate how our system will act in the real world and compares to similar frameworks. We would like to simulate a baseline dataflow computation to provide a comparison with other data flow systems. Furthermore, the development and contrast of a variety of load balancing and scaling mechanisms seems like useful, informative work.

Another fundamental piece of our study requires showing the ease of use of the programming abstraction. By developing an API for virtual actors, we hope to show that the programmability of the model is intuitive, and well suited to be adapted to cluster computing. Finally, no API alone is sufficient, and we would like to begin building a prototype of the system. We believe that the system building process will spawn new and exciting problems in this space that will motivate and guide our future research on these topics.

VI. Conclusion

In this paper we have presented the use of virtual actors as a distributed programming abstraction that may be a well suited alternative to the dataflow model of computation for variable, streaming data. By describing a computation as a number of actors, along with the communication flow between them, programmers may *easily define* a computational methodology that can *automatically* scale and load balance to handle higher rates of data. Generalized virtual actor spaces are high level abstractions that provide a rich low level of concurrency.

In order to test the model in the real world, we analyzed three applications and fit them to an actor cast. Through the process of application analysis we discovered three generic communication models that represent a variety of message passing patterns in streaming applications. We then implemented a discrete event simulation to test the utilization of actors through this variety of communications.

The Github repository with the code for our simulation can be found at **http://bit.ly/actors-simulation**.

ACKNOWLEDGMENT

We would like to thank Dr. Josh Rigler from USGS for discussing the magnetometer project with us, the available data sources and how it could be computed upon. We would also like to thank Dr. Michael Wiltberger from NCAR as well as Dr. Alan Sussman from UMD who put us in touch with Dr. Rigler. We'd like to thank Dr. Amol Deshpande for taking a look at our work and inspiring the initial actor model research. Finally, we'd like to thank Dr. Jeff Hollingsworth for his guidance and forbearance as we pursued this project as part of our High Performance Computing coursework.

REFERENCES

[1] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1327492

[2] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2. [Online]. Available: http://dl.acm.org/citation.cfm?id=2228301

[3] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters," in *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*. USENIX Association, 2012, pp. 10–10.

[4] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, and others, "Storm@ twitter," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 147–156.

[5] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernndez-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and others, "The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1792–1803, 2015. [Online]. Available: http://dl.acm.org/citation.cfm?id=2824076

[6] C. Hewitt, "Viewing control structures as patterns of passing messages," *Artificial intelligence*, vol. 8, no. 3, pp. 323–364, 1977.

[7] G. A. Agha, "Actors: A model of concurrent computation in distributed systems." DTIC Document, Tech. Rep., 1985. [Online]. Available: http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA157917

[8] J. E. Gonzalez, P. Bailis, M. I. Jordan, M. J. Franklin, J. M. Hellerstein, A. Ghodsi, and I. Stoica, "Asynchronous Complex Analytics in a Distributed Dataflow Architecture," *arXiv preprint arXiv:1510.07092*, 2015. [Online]. Available: http://arxiv.org/abs/1510.07092

[9] P. Haller and M. Odersky, "Scala actors: Unifying thread-based and event-based programming," *Theoretical Computer Science*, vol. 410, no. 2, pp. 202–220, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0304397508006695

[10] R. K. Karmani, A. Shali, and G. Agha, "Actor frameworks for the JVM platform: a comparative analysis," in *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*. ACM, 2009, pp. 11–20. [Online]. Available: http://dl.acm.org/citation.cfm?id=1596658

[11] P. Bernstein, S. Bykov, A. Geller, G. Kliot, and J. Thelin, "Orleans: distributed virtual actors for programmability and scalability," MSR Technical Report (MSR-TR-2014-41, 24). http://aka. ms/Ykyqft, Tech. Rep., 2014. [Online]. Available: http://131.107.65.14/pubs/210931/Orleans-MSR-TR-2014-41.pdf

[12] M. Research, "Microsoft Project Orleans Orleans Streams," 2015. [Online]. Available: http://dotnet.github.io/orleans/Orleans-Streams/

[13] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter Heron: Stream Processing at Scale," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 239–250.

[14] Y. Koren, "Collaborative filtering with temporal dynamics," *Communications of the ACM*, vol. 53, no. 4, pp. 89–97, 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1721677

[15] A. Gupta, G. Karypis, and V. Kumar, "Highly scalable parallel algorithms for sparse matrix factorization," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 8, no. 5, pp. 502–520, 1997. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=598277

[16] H.-F. Yu, C.-J. Hsieh, I. Dhillon, and others, "Scalable coordinate descent approaches to parallel matrix factorization for recommender systems," in *Data Mining (ICDM), 2012 IEEE 12th International Conference on*. IEEE, 2012, pp. 765–774. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6413853

[17] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis, "Large-scale matrix factorization with distributed stochastic gradient descent," in *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2011, pp. 69–77. [Online]. Available: http://dl.acm.org/citation.cfm?id=2020426

[18] J. J. Love and C. A. Finn, "The USGS Geomagnetism Program and its role in space weather monitoring," *Space Weather*, vol. 9, no. 7, 2011. [Online]. Available: http://onlinelibrary.wiley.com/doi/10.1029/2011SW000684/full

[19] D. J. Hill, B. S. Minsker, and E. Amir, "Real-time Bayesian anomaly detection for environmental sensor data," in *Proceedings of the Congress-International Association for Hydraulic Research*, vol. 32. Citeseer, 2007, p. 503. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.329.8262&rep=rep1&type=pdf

[20] M. S. Wheatland, "A Bayesian approach to solar flare prediction," *The Astrophysical Journal*, vol. 609, no. 2, p. 1134, 2004. [Online]. Available: http://iopscience.iop.org/0004-637X/609/2/1134

[21] R. Qahwaji and T. Colak, "Automatic short-term solar flare prediction using machine learning and sunspot associations," *Solar Physics*, vol. 241, no. 1, pp. 195–211, 2007. [Online]. Available: http://link.springer.com/article/10.1007/s11207-006-0272-5

[22] D. Yu, X. Huang, H. Wang, and Y. Cui, "Short-term solar flare prediction using a sequential supervised learning method," *Solar Physics*, vol. 255, no. 1, pp. 91–105, 2009. [Online]. Available: http://link.springer.com/article/10.1007/s11207-009-9318-9

[23] D. Yu, X. Huang, Q. Hu, R. Zhou, H. Wang, and Y. Cui, "Short-term solar flare prediction using multiresolution predictors," *The Astrophysical Journal*, vol. 709, no. 1, p. 321, 2010. [Online]. Available: http://iopscience.iop.org/0004-637X/709/1/321

[24] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le, and others, "Large scale distributed deep networks," in *Advances in Neural Information Processing Systems*, 2012, pp. 1223–1231. [Online]. Available: http://papers.nips.cc/paper/4687-large-scale-distributed-deep-networks

[25] N. Matloff, "Introduction to discrete-event simulation and the simpy language," *Davis, CA. Dept of Computer Science. University of California at Davis. Retrieved on August*, vol. 2, p. 2009, 2008. [Online]. Available: http://web.cs.ucdavis.edu/~matloff/matloff/public_html/SimCourse/PLN/DESimIntro.pdf