Distributed State Machines: A Declarative Framework for the Management of Distributed Systems

Jorge Lobo, David Wood, Dinesh Verma and Seraphin Calo

IBM Research
Hawthorne, NY, USA
jlobo,dawood,dverma,scalo@us.ibm.com

***Abstract*— In this paper we describe the implementation of a declarative framework to support the development of distributed management applications. The framework is based on an extension of declarative networking, an asynchronous computational model that uses recursive SQL as its foundation and has been successfully used for the implementation of multiple networking protocols including opinion-based preferential routing as well as standard path vector and link state routing protocols. The SQL implementation enables analysis capabilities that can help avoid implementation and logic errors.**

I. Introduction

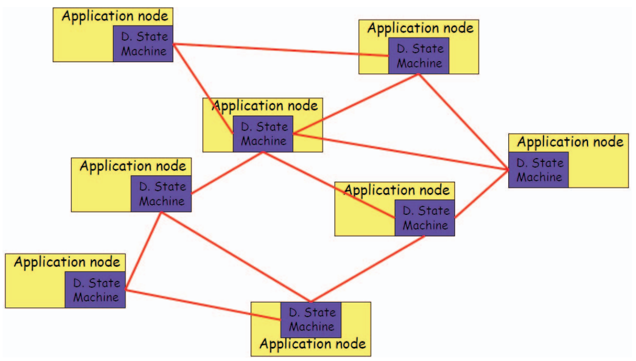
Over the past year and one half we have been building on an infrastructure to support the development of networking applications. The goal is to relieve the application developer of tending to the details of the communication and control. The work has been motivated by the congruence of the need for such an infrastructure in several areas of research in IBM: collaboration of sensor nodes and other data providers across hybrid networks to produce actionable information and knowledge for a Smarter Planet; monitoring and management of workloads in the cloud; management and distribution of workloads to the edge of a Mobile Network Operator's hierarchical network to minimize network bandwidth requirements; enablement of malleable mobile ad hoc networks with facilities for network elements to work together seamlessly towards a common goal.

Distributed decentralized applications that are robust to changing relationships and topologies, with no single points of failure are difficult to write and we would like to abstract out as much of the networking as possible. We take a data-driven model of computation in which computational application nodes communicate with each other through the exchange of data to solve a distributed problem. Data is collected locally and shared with others as defined by the application. Changes in local data shared with other peer application nodes can trigger those peers to take appropriate action in response to these changes. The framework is composed of three layers. The lowest layer defines the concept of a Node System. The Node System specifies the basic characteristics of the topology where applications will be deployed. From the application perspective that is installed in a node the Node System provides a specific set of operations (or relationships) to identify other nodes. For example, a mesh Node System allows the application to identify all the neighbors of the node. In a tree Node System, the application will be able to identify the parent node and the child nodes, and maybe all the ancestors or descendents, if the Node System supports such operations. The Node System concept puts these various relationships in a common API. An application can directly share data with peers that are accessible through the Node System. The framework is in charge of efficiently managing the data shared between the nodes. The second layer supports the concept of shared data. This concept is implemented using a set of APIs available to applications to specify the data that they want to share and methods to set call-backs when some specific condition over the local data and data shared by other nodes become true. The framework is also in charge of monitoring the local and remote data and making the appropriate callbacks when the application defined conditions are met. The third layer is related to the nature of the data that can be shared, and provides different models for data sharing. We will briefly discuss two models, one based on serializable Java objects and conditions on these objects and the other based on relational databases and conditions described in an SQL-based language.

The rest of the paper is organized as follows. In the next section we will describe in more detail the Node System and the APIs. In Section III we describe the Java based data sharing model and in Section IV we describe the SQL-based sharing model and the language to support it. Related work is described in Section V and some final remarks and future direction for the project are presented in Section VI.

II. DSM Architecture

The Distributed State Machine architecture assumes a simple model of a distributed set of application nodes, each maintaining its own local data, and collaborating to solve a common problem by sharing data as necessary. An application will be distributed among different nodes and communication between nodes will happen through the Distributed State machines.



Instead of directly addressing nodes, an application uses network topology relationships such as parent, neighbor, reachable, etc., to address nodes. These features enable, for example, the implementation of routing algorithms in a mobile ad hoc network through the sharing of neighbors and known routes with node neighbors. With this framework in mind, the primary goals of the architecture are to: 1) provide topology-based addressing; 2) support dynamic networks; and 3) allow the efficient and intuitive sharing of data.

The high level architecture of a DSM node is depicted in Figure 1. A single hop broadcast/multicast algorithm is implemented within each node with both nodes and routes assigned an expiration time. This provides reachability and neighbor information about other nodes and also allows the network to partition and adapt to the changing set of connected nodes. Logical topology relationships such as *parent* and *child* are defined by the application through a platform configuration. The topological relationships of the nodes are made available to the rule system to allow addressing of messages based on topology.

Shared data is assigned a unique name defined by the application. The data is then made available to peers via queries that reference one or more named data objects. Queries are either synchronous or continuous and are addressed to a set of nodes with one or more topological relationships (i.e. neighbor, parent, etc.). For example, a node might request the resources for each virtual machine running on its children (assuming the nodes are sharing such information). Continuous queries may specify a condition that must be met before results are returned. Conditions may be defined on a pair-wise (peer-to-peer) basis or as an aggregation of values across multiple nodes in a topological relationship. For example,

- pair-wise: return the virtual machine resources for those nodes that are exceeding 50% cpu utilization.
- aggregation : return results when the local node's cpu utilization is less than the average of all neighbors.

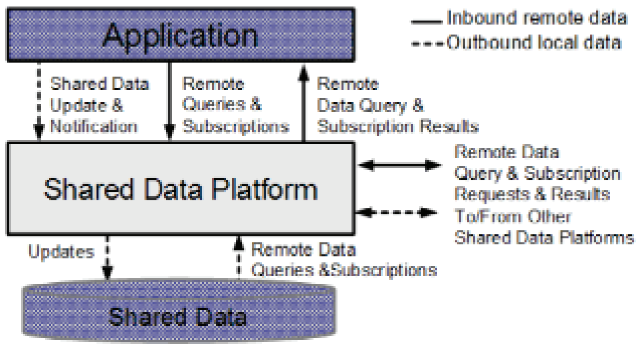

Application
Inbound remote data
Outbound local data
Shared Data Update & Notification
Remote Queries & Subscriptions
Remote Data Query & Subscription Results
Shared Data Platform
Remote Data Query & Subscription Requests & Results To/From Other Shared Data Platforms
Updates
Remote Data Queries &Subscriptions
Shared Data

Figure 1 Architecture of a DSM node

The application development progresses in three-stages. (1) A System Developer extends or configures a Node System. The Node System implements a set of relationships among nodes and supports specific types of operations and relations on remote data. (2) an Application Developer defines an application using the Node System to register local data to be shared with peers., and may register callbacks when some conditions hold true on data in related nodes; and, (3) the application is deployed on one or more physical nodes.

To accommodate a broad set of application domains, the framework is designed to support generic shared data types. Currently, the architecture has been implemented using two distinct shared data types: one for serializable Java objects and the other for rows (tuples) in database tables. We briefly describe features of the Java implementation and then we describe the details of the implementation of a Datalog-based language on top of the tuple implementation.

III. SHARING JAVA OBJECTS

The implementation for sharing serializable Java objects uses a hash table to map the names associated with shared data to the shared Java instance data. Support for queries based on the names of shared items is supported and more complex queries, such as for members of a hash table, can easily be added. A powerful aspect of this implementation is the availability of the Java Expression Language JEXL based expression to define the conditions used in continuous queries. The JEXL run-time is provided the following set of objects, over which expressions can be defined:

- ***Local*** - a hash table of shared Java instances from the local node
- ***Peer*** - a hash table of shared Java instances from a single remote application peer
- ***Average*** - provides average values of Java Number values from 1 or more remote peers
- ***Sum***- provides the sum of values of Java Number values from 1 or more remote peers
- ***Min***- provides the minimum of values of Java Number values from 1 or more remote peers
- ***Max***- provides the maximum of values of Java Number values from 1 or more remote peers, and
- ***Count*** - provides the count of named remote Java instances

The *Local* and *Peer* instances' *get()* method can be used to acquire the Java instances and define expressions using them. The aggregator instances (*Average*, *Sum*, *Min*, *Max* and *Count*) provide the methods:

- Number get(String dataName, String memberName);
- Number get(String dataName);

where *memberName* may be a the name of a field within a Java object or a zero-args method that evaluates or can be coerced to a Number object. The only restriction is that the *Peer* instance and aggregator instances can not be used in the same expression. Some examples over data shared with the name *resources* follow:

- Local.get("resources").cpuUtilization < Peer.get("resources").cpuUtilization
- Local.get("resources").cpuCount() < Average.get("resources","cpuCount()")
- Count.get("resources") > 4

These operate on a shared Java object containing 1) a field named *cpuUtilization* that is either a java.lang.Number or a primitive number; and 2) a method *cpuCount()* that returns either a Number or primitive number. The condition in the first example will be come true whenever the cpu utilization of a peer node is larger than the cpu utilization of the current node. The second condition will become true whenever the average number of cpu's in all the peer nodes is larger than the number of cpu's in the local node. All the communication required among the nodes to compute these conditions is transparent to the application developer.

IV. Sharing Relational Tables

The computation based on the relational model is very simple. Each node runs an input/output state machine. A state in the machine is represented by set of relational tables. State transitions occur when the state machine receives inputs either from applications running in the node or from other state machines running in other nodes (i.e. communications between nodes only happens through the state machines). All inputs are in the form of named tuples (i.e. relational table rows). A system-defined input table provides a list of nodes and their topological relationships thereby enabling the addressing of messages based on topology. A state change can also produce an output in the form of tuples, all output tuples are accompanied with a destination, i.e. the location of another state machine which will asynchronously receive the tuple. As a simple illustration, assume we are collecting load data as part of the implementation of the distributed load balancing algorithm described in [2]. In this algorithm, a node decides to offload or accept from another node some of its virtual servers based on its load, the load of its neighboring nodes and the latency of communication with those nodes. A typical state of a node can be represented by the following table:

Loads

NodeId	Load	Capacity	Latency
"n1"	350	1200	3230
"n2"	100	1200	2900
"n3"	700	3500	2000

Changes in this table (i.e. changes of state) can happen when a new neighbor sends new load information. The state transition function is defined using rules similar to Datalog [8]. There are three types of rules that are used to define the function. The first type of rule defines insertion of tuples into the state. In our example a rule to insert tuples into the Load table can look like the following:

```
load(ID,Load,Capacity, CurrentTime -TimeRe) +=
        loadInfo(Load,Capacity,TimeReq)@ID,
        get_current_time(CurrentTime);
```

In this rule, *loadInfo(Load,Capacity,TimeReq)* is a tuple schema of a tuple received by the local node that was sent from the node *ID* (*ID* in the schema is a variable that gets its value when the tuple arrives to the local node, similarly *Load, Capacity, TimeReq* are variables that get their values from the arriving tuple). The *get_current_time* table is a built-in table that returns the current time each time that is queried.

The second type of rule removes rows or tuples from the tables. In our example, we can have a rule of the form:

```
load(ID,Load,Capacity,Latency) -=
        loadInfo(Load,Capacity,Latency)@ID;
```

This rule removes from the **Load** table tuples with node id *ID* if a *loadInfo* tuple is received from *ID.* To distinguish tuples that can be part of the state and tuples that are sent between nodes we call the former tuples *state tuple* and the latter *transport tuples.*

The last type of rule defines when transport tuples are sent. In our example, when to send *loadinfo* tuples is defined as follows:

```
loadInf(Load,Capacity,TimeReq)@Req +=
        requestLoad(TimeReq)@Req,
        localLoadCapacity(Load,Capacity);
```

Load information is sent when it is requested by a node through a transport tuple *requestLoad.* This transport tuple comes with the time when the tuple was sent out. This value is returned together with the local load and capacity to the requester. Note that *@Req* in the *requestLoad* tuple indicates the ID of the node the tuple is coming from, but in the *loadInf* it indicates where the tuple will be sent to. A request can be sent out periodically. This can be specified by the rule:

```
requestInfo(CurrentTime)@ID +=
        get_current_time(CurrentTime),
        neighbor(ID): MSEC = 10000;
```

The annotation in the rule limits its evaluation to every 10,000 milliseconds. This will broadcast a *requestInfo* tuple to all the neighbors of the node which are maintained in the table *neighbor* and stored as part of the state. In [2], load balancing decisions are based on the local utilization which really corresponds to the utilization in the neighborhood. This computed by adding the load in the node plus the load in all the neighbors divided by the sum of their capacities. This value can be maintained using rules as follows:

```
local_capacity(SUM(Cap)) :=
        loads(*,*,Cap,*);
```

```
local_load(SUM(Load)) :=
        loads(*,Load,*,*);
local_utility((LL + L) /( LC +C))  +=
        local_capacity(LC), local_load(LL),
        node_load_and_capacity(L,C);
```

In these rules we introduce some new notation: *aggregation*, *transient tables* and *input tables*. Aggregation, as in relational databases, aggregates values in a column of a table into a single result. In the first rule, the values in the third column of the *Load* table, the capacity of the neighboring nodes, are added up and stored in a new table named *local_capacity*. Note also that this rule uses *':='* instead of *'+='* to denote where the result of the condition will be stored. This distinction is made because the *local_capacity* table is not part of the node state. It is only a temporary table used in the third rule that generates the local utility which will be part of the node state. Similarly, the second rule is generating a temporary table to store the local load. These two tables are dropped after the evaluation of the rules is completed. The *local_capacity* and *local_load* are, thus, called *transient tables*. Finally, the *node_load_and_capacity* table is a table created by the hosting application in the node and input to the DSM. The application should periodically get the load of the node and pass it to the DSM together with the node's capacity. This input will trigger the evaluation of all the rules. Tables which are created and input to the DSM by the application are called *input tables*. Similar to transient tables, input tables are not part of the DSM state and they will be dropped after the evaluation of the rules are completed. Every rule set comes accompanied by a declaration of tables and input tables are recognized by their declaration. In our application we have:

```
input node_load_and_capacity(int L; Int C);
transient local_capacity(int C);
transient local_load(int L);
state load(char[120] ID; int L; int C; long Let);
state local_utility(int U);
state localLoadCapcity(int L; int C);
state neighbor(char[120] ID);
transport requestLoad(long T);
transport loadInfo(int L; int C; long T)
```

There are well known algorithms to translate Datalog rules into SQL [8]. The special case in our language is the manipulation of transport tables. When a transport table is mentioned in the head of rule, tuples generated by the rules are added to the table and the table is shared with the peer nodes using the Node System. If a transport table is mentioned in the body of a rule, the rule will be evaluated when a transport tuple is received from one of the peers and inserted into the table. This notification is implemented using a Node System's continuous query.

V. RELATED WORK AND FINAL REMARKS

Our work on distributed state machines is an extension to the concept of declarative networking introduced in [5]. Declarative networking platforms have been developed specifically to support the declarative implementation (Datalog based) of routing protocols. However, the computational model behind declarative networking is directed towards the definition of a single state: a stable state that the system is intended to reach. State transition happens but they are supposed to be hidden from the application developer. In our implementation we follow the suggestion made in [1] to explicitly represent state. There are important semantic differences between our implementation and the implementation in [1] but details of that differences are outside the scope of our presentation. Details can be found in [3]. Another important contribution of our system that is not presented in previous implementations is the concept of Node Systems. In previous systems the network of nodes where the distributed application was supposed to run was never considered. Having an infrastructure for nodes systems simplifies the deployment of applications. Building a particular instance of a Node System is not trivial, but this can be done once and use many times by any applications.

We are in the early deployment stages. We are using our framework to build applications that provide distributed execution and control of video analytics in a surveillance context, tracking of assets in a sensor field, fault management and resiliency in networked appliances, and for management of resources in a data center environment. The framework can be used to simplify the development of many new applications, primarily in the field of network and systems management, and in processing of sensor network information streams. The results are anecdotal but we have done some testing developing a universal proxy for P2P file sharing protocols, and although the distributed parts of the proxy are limited the versatility of the declarative rule language allows us to quickly adapt the proxy to multiple protocols and easily reconfigure the proxy to support either caching or access control. Evidence of similar languages for the management of network routing shows that developing and maintaining network protocols are significantly simplified [6] using this approach and we expect similar results using our system, but time will say.

An important aspect of the declarative approach is the amenability for analysis. We are developing a toolkit for debugging and analysis. We currently have a basic set of analysis tools [7] and we have used it to check properties of three different network routing protocols: BGP, a generic link state protocol and a MANET protocol. For BGP, we addressed a question that has been shown to be difficult to answer: given a network configuration, potentially with a dispute wheel [3], does the network always converge? In the link state protocol we used the tool to identify the presence of persistent forwarding loops, and in the MANET protocol we analysis problems related to the discovery of disjoint paths.

ACKNOWLEDGMENT

Research was sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defence under Agreement Number W911NF-06-3-0001. The views and conclusions are those of the authors and do not represent the U.S. Army Research Laboratory, the U.S. Government, the U.K. Ministry of Defence or the U.K. Government. The U.S. and U.K. Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding copyright notation.

References

[1] P. Alvaro, W. R. Marczak, N. Conway, J. Hellerstein, D. Maier, and R. Sears, ‘Dedalus: Datalog in time and space’, in Datalog, pp. 262–281, (2010).

[2] Zhenyu Li, Gaogang Xie: A Distributed Load Balancing Algorithm for Structured P2P Systems. ISCC 2006: 417-422

[3] T. G. Griffin, F. B. Shepherd, and G. Wilfong, “The stable paths problem and interdomain routing,” in IEEE/ACM Transactions on Networking, 2002.

[4] J. Lobo, J. Ma, A. Russo and F. Le. Declarative Distributed Computing. In E. Erdem et al. editors, Lifschitz Festschrift, LNCS, 2012, Volume 7265/2012, 454-470.

[5] B.T. Loo. The Design and Implementation of Declarative Networks. PhD thesis, EECS Dept., UCBerkeley (Dec. 2006).

[6] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. 2009. Declarative networking. Commun. ACM 52, 11 (November 2009), 87-95

[7] J. Ma, J. Lobo, F. Le, A. Russo. A Declarative Fremework for the Analysis of Network Protocols. Under review.

[8] Jeffrey D. Ullman: Principles of Database and Knowledge-Base Systems, Volume II Computer Science Press 1989