CoorSet: A Development Environment for Associatively Coordinated Components

Kevin Kane and J. C. Browne

Department of Computer Sciences, The University of Texas at Austin
1 University Station C0500, Austin, TX 78712-0233 USA

{kane, browne}@cs.utexas.edu

Abstract. A development environment for applications specified in an extended version of a previously developed coordination model based on associatively broadcast interactions is presented. The previous associative broadcast coordination model is extended to incorporate more complex specifications for interactions including multiple message interactions and fault-tolerance by replication. The runtime system is extended to facilitate construction and application of distributed implementation of coordination systems. An interface definition language based on the extended coordination model and a compiler for the language are defined and described. Three example applications, a generalized readers/writers problem including replication, a "greedy reuse" algorithm and a distributed computation Google pageranks are presented.

1 Introduction

There has been little research on coordination models and languages based on broadcast communication, despite the fact that many network intrinsically provide a physical broadcast capability including such widely available systems as Ethernet, FDDI, and wireless, and that broadcast enables consensus for asynchronous communication [2].

There has also been, except for Linda-based [5] coordination models and languages, relatively little experimental or systems-oriented research on application of coordination models and languages. Experimental research is needed to establish a basis for application of coordination models and languages and to add credibility to the utility value of coordination models and languages. This paper extends previous research on coordination models based on associative broadcast to a development environment for implementation of coordinating systems of processes, illustrates its applications and positions this research in the context of distributed and peer to peer systems research.

The goal for the development environment is to facilitate experimental research on broadcast-based coordination systems. The principal artifacts of the development environment are: extensions to the previous associatively broadcast programming model to facilitate experiments and applications, an interface definition language for

expressing associative interactions, a compiler for this interface definition language and an environment for instantiating and executing coordinating systems of processes.

Broadcast enables coordination based on every process in an interacting set locally maintaining common state necessary for collective decision procedures [1, 2]. Associative broadcast [3, 4] enables targeting of messages to processes in specific states and enables each process to select the properties of messages it will receive. Basing coordination on associative broadcast communication enables definition of multiple dynamic coordination subsets in a set of processes. Separation of message filtering from computation decreases the execution cost of coordination using broadcast and allows for specialization to specific algorithm requirements. Associative broadcast preserves anonymity similarly to tuple space based coordination [5]. It enables transparent distribution and replication for fault-tolerance. In summary, associative broadcast enables fully distributed and fully symmetric coordination over dynamic sets of processes.

The next section summarizes the results from previous research on definition and implementation of a coordination model based on associative broadcast. Section 3 sketches the extended coordination model. Section 4 discusses formulation of algorithms as coordination systems in the extended coordination model. Section 5 sketches the CoorSet interface definition language and illustrates formulation of algorithms and computations in the CoorSet coordination development environment including discussion of definition of algorithms. Section 6 sketches the environment for instantiating and executing systems of coordinating processes. Section 7 discusses the implementation of our system. Section 8 gives related research and section 9 gives a summary and conclusions.

2 Associative Broadcast Coordination Model

A previous paper [6] reported a coordination model based on extending communication by associative broadcast into a coordination model by extending associative communication to associative interactions. This coordination model will be referred to as Associatively Specified Interactions (ASI). ASI is a model of coordination amongst a group of *components*. A *component* is a logically distinct process executing on some host in a network. Components may coexist with other components on the same physical host, or each may reside on a different host. The ASI implementation of a component is one or more functions encapsulated by an interface which implements the ASI interaction protocols. In the ASI protocols, a target set specification travels with each message that is broadcast onto the network. The target set is determined for each message by the recipients whose local state satisfies the target specification. The sender does not know the membership of this set, and does not necessarily ever discover it.

The state of a component is specified in the interface as a "profile," which specifies the visible current state of the component. Profiles are implemented as sets of attribute/value pairs taken from the attribute domain of the component set that specify a *descriptive name* [3] for the component. The target set for a message is determined by a conditional expression called a "selector," which is a predicate evaluated against

the profile of each component. The message is broadcast throughout the network, but is received only by those components for whose profiles the selector evaluates to true. This allows targeting of messages to subsets of components that have a desired state, without the sender knowing the membership of that set. All components in the system configure their profiles and broadcast messages with selectors according to a coordination protocol. Protocols implementing acknowledgements can be implemented if needed for a given coordination problem.

3 Extended Associative Broadcast Coordination Model and Programming Model

The previous associative broadcast coordination model has been extended into a programming model which enables direct representation of complex interactions with retention of separation of concerns. This model incorporates two additional features: complex conditions for enabling execution of a component and replication for both representation of SPMD parallelism and fault-tolerance.

The conditions for executing and action of a component commonly include receipt of multiple messages. To maintain separation of concerns it is necessary to incorporate this requirement into the coordination model.[1] We introduce the concept of a "firing rule" into the coordination model. A firing rule is a specification of the set of messages which must be received to initiate any action of a component. Additionally, since components may and often will have persistent state, there may be precedence relations among possible enabling message sequences. These extensions are accomplished by adding types to messages and incorporating a conditional expression over message types and local state into the associative interface.

The definition of firing rules used in the extended coordination model is taken from a data flow programming model [7], where rather than waiting on a single input, a node in a data flow graph waits on multiple inputs, possibly in a particular order, before becoming enabled for execution. Firing rules are specified with a Java-like logical syntax. Specifying reception of *either* of two message types *R, S* is done with a rule "R || S". Reception of *both* of two message types is specified with a rule "R && S". Reception of *R followed by S* is specified with a rule "R < S". These rules can be compounded and grouped with parentheses, such as "(R < S) || (R < T)". The '<' operator has the lowest precedence, followed by `||', and `&&' has the highest precedence.

Replication is another feature that must be included in associative interaction specifications to enable facile specification of parallelism and fault-tolerance. SPMD parallelism can be readily implemented by replication of components. Replication of functionality for fault-tolerance can be made transparent and synchronization-free after initialization. If an initiating component starts several replicas of a given component to insure success in an unreliable environment and each of the replicated

[1] In the previous coordination model, if multiple messages were required to enable an action by a component, the set of actions of the component had to include aggregation of these messages in effect breaking separation of concerns.

components responses by associative broadcast then the initiating component can safely proceed after the first successful result and set its profile to ignore the other completions. A component can be replicated by adding an index attribute to its profile and instantiating replicas in conformance to the index range. Once specified, a component can be started an arbitrary number of times. The runtime system will provide unique identifiers in a predictable way so replicas can alter their behavior, or they can all execute in the same way depending on the needs of the application.

A component in the extended model is a 5-tuple *(S, S_0, P, A, R)* where *S* is the state machine which implements the rules for the protocol specification, and the rules for profile and interface changes, S_0 is the initial state, *P* is the profile of attributes and attribute-value pairs, *A* is the list of accepted transactions (*T*, T_A) where *T* is a firing rule and T_A is the argument signature, and *R* is the list of requested transactions (*T*, T_A) where *T* is a transaction type and T_A is the argument signature. Section 5 illustrates the concepts in the extended model in CoorSet language examples.

4 Algorithm Formulation

Most distributed algorithms explicitly or implicitly are formulated on the assumption of central control. Coordination models, on the other hand, do not assume central control. Development of algorithms and computations in coordination models therefore requires a shift in development paradigm. Use of a coordination model based on broadcast communication induces a further shift in development paradigm since most distributed computations and coordination models are based on point to point communication.[2] There has been relatively little research in formulation of distributed/parallel algorithms in broadcast models of computation [8].

The development paradigm for distributed algorithms formulated in associative interactions is the integration of component composition and component interactions. An algorithm is specified as coordination among a set of components. Composition defines the structural relationships among components while coordination specifies the behavior of the composed system. Associative interactions use the same representation to specify both coordination and composition.

A coordination system is implementing an algorithm or computation is specified in terms of a set of attributes in which the profiles and selectors are specified, a set of components from which the algorithm or computation can be composed, a set of protocols in which interactions are specified including message types, the selectors to accompany each instance of a message type, the allowed sequences of messages and the responses to each instance of a message type which is received, and a state machine which implements the coordination protocols which are interfaced to each process or component.

[2] Linda-based coordination models [5] are the exceptions to this generalization.

5 CoorSet Interface Definition Language

CoorSet is an interface definition language for specification of the behaviors of components in terms of associative interactions. The CoorSet compiler generates Java code to implement the coordination models for each component and a "main" component that starts the application in the runtime system described in Section 6. In the example that follows, components of the language that deal with details not directly related to the interface structure have been omitted for clarity; for complete details of the language, see [9].

```
component 5 {
 profile ("ReaderWriter", ("EID", 2),
        ("Status", "initializing"))

 execute startUp

 accepts "RequestToRead" processRead ()
 accepts "RequestToWrite" processWrite (Object)

 rule "update_value < (update_done || Collision)" processUpdate

 requests "ReplyFromData" sendReply "Client" (Object)
 requests "update_value" attemptUpdate "ReaderWriter" (Object, Integer)
 requests "update_done" completeUpdate "ReaderWriter" (Integer)
 requests "Collision" updateCollide "ReaderWriter" (Integer)
}
```

Fig. 1. CoorSet definition of a replicated data object store.

5.1 Readers/Writers Algorithm In CoorSet

To demonstrate the language for describing components, we introduce a generalized distributed readers/writers system implemented in CoorSet. The data objects are replicated for fault-tolerance. Consistency is maintained across non-malicious failures of components and/or runtime creation of additional replicas. This generalized readers/writers problem is rather complex when programmed in conventional distributed/parallel programming languages but is quite simple in CoorSet. The readers/writers system consists of a set of reader/writer objects which store a data item that is replicated across multiple independent stores, and a set of client components which randomly invoke reads and writes of randomly selected data items. Each reader/writer is a single component in the system. Each encapsulates and stores a single replica of a single data item, provides reading and writing facilities to clients, and implements a coordination protocol amongst all the other replicas of that

data item when a write is requested. Each replica's profile contains the unique identifier of the data object, and an index to indicate which replica it is. Each reader/writer component keeps track of the version number of its data item, increasing it each time an update is made. When two updates are attempted simultaneously, meaning they are sent out with the same sequence number, they are said to "collide." When they do, they are aborted. Each then executes an exponential backoff algorithm before attempting the update again with a new sequence number. A definition fragment for the reader/writer component is given in Figure 1.

The "5" immediately following the "component" declaration specifies that five instances will be started of a component which has an initial profile of three entries. First, an ID attribute named ReaderWriter distinguishes it from other kinds of components in the system, such as clients. Second, a valued attribute EID (for entry identifier) indicates which data item this object contains. Here the entry identifier is declared in the configuration file, implying that each data object's replicas are declared separately. Third, a valued attribute named Status with the value initializing. This is the state of the component when it initially comes online, to show that it is not yet operational, and needs to synchronize with whatever other data stores are already in operation. This attribute later changes to values local processing, reading, and writing to reflect the various states it is in when processing requests.

This component type accepts two message types, RequestToRead and RequestToWrite. These requests are made by clients who want to read and write the data item, respectively. In each case, reception of these messages causes execution of the methods processRead and processWrite on the programmer-supplied computational code (not shown), each of which takes the given parameter types.

The readers/writers component also implements a firing rule which first receives an update_value message, and then an update_done message to indicate the update is successful, or a collision to indicate two writes were attempted at the same time and collided.

This component also requests four message types. ReplyFromData is the response sent to clients in response to a request to read or write. It carries a single Object parameter, which will contain a copy of the data item when read. Its default selector of "Client" will be received by all clients, but when such a response is actually sent, the selector will be refined to target only the requesting client. update_value, update_done, and Collision are all sent during the various stages of consensus to attempt an update of the data item, to signal the update is successful, or conversely, when two attempted updates collide and must be aborted. Each takes the new sequence number of the updated data, and for update_value, the new data value. Each of these are by default targeted to all other data stores by the default selector "ReaderWriter". The optional execute line specifies a method on the programmer-supplied class to execute immediately upon component start-up; if this line is absent, the component just waits for incoming transactions upon starting.

For a simple performance study the data objects were replicated 2 and 4 times. The number of processes reading and writing was varied up to 64, each on a separate workstation on a network. The average number of messages was about Nx2.5 where N is the number of *data object* replicas. Note that the performance of the algorithm is almost independent of the number of readers and writers.

5.2 Data Fitting Example

We now present a more complex example of distributed data fitting, motivated by the concept of "greedy reuse" [10]. "Greedy reuse" uses execution of multiple, perhaps redundant components, to ensure the success of a computation by simultaneously executing multiple implementations of a required functionality when it is not certain which implementation should be used. "Greedy reuse" is complex to program in conventional distributed programming systems but simple as a coordination language program. Consider an application that collects a set of data points, and requires approximating them by a curve. There are many possible approaches to data fitting. Consider for illustration a case where it is unclear simply from the data set what method will yield a fit with certain properties required by the application. Possible properties are a minimum of error, compactness of representation, and smoothness of curve. It may be that the requirement is satisfied only by a composition of fits.

Using associatively-coordinated components, several data fits can be executed simultaneously by addressing a data set with a selector that matches to true for the profiles of all data fitting components. The selector can be made more specific if only certain types of fits are desired. This application has more obvious connections between components as is common in more typical coordination models, and the component which initiates the computation is separate from the one that receives the result, to give a linear data flow as illustrated by Figure 2. There is no explicit link between these components, and each circle in fact represents any number of components of that type which may be operating when the request is made. These links should be seen as dynamic, existing only as long as they are required.

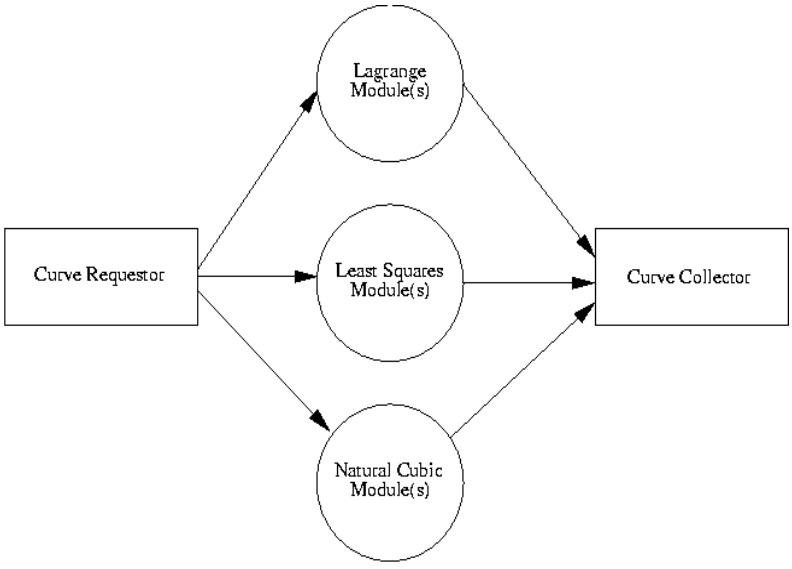

Fig. 2. Data flow between components of the data fitter application.

Transparent replication and fault-tolerance is obtained by having several copies of the same type of component running, and when the calling component receives all of the results, it can compare them to choose the ones which meet the requirements, or alternatively, those which are faulty. In an unreliable environment the initiating component might choose to simultaneously execute multiple copies of another component just to insure that a result is computed and successfully received with high probability.

We have, for this illustration of concepts, implemented components that provide an exact Lagrange interpolating polynomial fit, a least squares approximation, and a natural cubic spline fit. Each component maintains a profile that identifies it as a data fitting component for the purposes of addressing, as well as profile entries that allow it to be addressed more directly when an application wants only a particular kind of fitting.

The dynamic structure of associative broadcast allows an application to link to all components available at the time a request is made, and to do so without explicit knowledge of what components are available; simply the knowledge of the *accepts* interfaces used by data fitting components is sufficient.

There are two possible system configurations for the components. The components can be active as daemons on hosts in the broadcast network in which case the initiating component is invoked on some hosts and discovery, linking and execution proceeds as previously described. Alternatively, the components can be in a file with the initiating component or in a library. In these latter cases the runtime system will distribute the components to hosts in the broadcast network and start the associative interaction runtime system.

Figure 3 contains the definition fragments for each of the types of components in CoorSet. These interface definitions are used to create the initial configuration of the component network. Some details have been omitted due to space constraints.

In this example, we have five types of components. In this case, three of each of the fitter components is started, as indicated by the "3" after the "node" keyword, to create replicated instances. There is only a single instance of the other components, CurveRequestor and CurveCollector. The types of components are:

- *CurveRequestor*: A component that has collected some data set, and requires it be fit to a curve. It does not accept any transactions, but makes a "DataFit" request to all data fitting components by way of its selector.
- *LagrangeModule*: An exact Lagrange interpolating polynomial. Its initial profile has one attribute called "DataFitter" to indicate that it is a data fitting component, and a valued attribute called "Method" with a value of "Lagrange" to specify the particular kind of data fitter it is. It accepts the "FitData" request, and makes a "FitDataRespnse_Poly" request to send its result.
- *LeastSquaresModule*: A least squares polynomial fitter. Its interface is almost identical to that of the Lagrange module, except that its profile reflects its being a Least Squares fitter, rather than a Lagrange interpolating polynomial fitter.
- *NatCubicModule*: A natural cubic spline fitter. It accepts the same "FitData" transaction, but responds with a "FitDataResponse_Spline" transaction, which contains a spline rather than a single polynomial.

- *CurveCollector*: A component that accepts the resulting curve fits from the above components. Its default profile contains an attribute called "CurveCollector," which also is the default for selectors for the responses from the above components.

There are also three types of requests for service:

- *FitData*: A request for a data fit. This transaction has four parameters: the

```
component 1 {
  class CurveRequestor
  execute start
  requests "FitData" Request "DataFitter" (String, Double[], Double[], Integer)
}
component 3 {
  class LagrangeModule
  profile ("DataFitter", ("Method", "Lagrange"))
  accepts "FitData" processRequest (String, Double[], Double[], Integer)
  requests "FitDataResponse_Poly" sendResponse "CurveCollector"
            (String, Double[])
}
component 3 {
  class LeastSquaresModule
  profile ("DataFitter", ("Method", "LeastSquares"))
  accepts "FitData" processRequest (String, Double[], Double[], Integer)
  requests "FitDataResponse_Poly" sendResponse "CurveCollector"
             (String, Double[])
}
component 3 {
  class NatCubicModule
  profile ("DataFitter", ("Method", "NatCubicSpline"))
  accepts "FitData" processRequest (String, Double[], Double[], Integer)
  requests "FitDataResponse_Spline" sendResponse "CurveCollector"
            (String, Cubic[], Cubic[])
}
component 1 {
  class CurveCollector
  profile ("CurveCollector")
  accepts "FitDataResponse_Poly" processPoly (String, Double[])
  accepts "FitDataResponse_Spline" processSpline (String, Cubic[], Cubic[])
}
```

Fig. 3. Initial configuration of data fitting in *CoorSet*

first, a String, specifies a transaction identifier, so that multiple fits may be requested and the responses can be connected with the appropriate request. The next two parameters are arrays of Double values, representing the X and

Y coordinates of the data points. The final Integer parameter specifies the maximum order of the polynomial, for polynomial fitters that can bound the polynomial degree.

- *FitDataResponse_Poly*: A response to a data fitting request, containing a polynomial fitting to the data. It carries a String with the transaction identifier for which this is a fit, and an array of Double values representing the polynomial coefficients.
- *FitDataResponse_Spline*: A response to a data fitting request, containing a natural cubic spline fitting to the data. It also carries a String transaction identifier, as well as two arrays of Cubic polynomials. The first is a piecewise parameterized representation of the X coordinates of the spline, and the second is a piecewise parameterized representation of the Y coordinates.

5.3 Distributed Computation of Google Pageranks

The Google PageRank algorithm [31] is the computation of the eigenvectors of the lowest eigenvalue of a matrix defined by the link structure of web pages. This computation is readily formulated in asynchronous iteration [32]. We have implemented a distributed, dynamic computation [27] of pageranks in CoorSet. Documents which have url-like links are distributed across a set of hosts (which model web servers.) coupled by a broadcast capability. The pageranks are computed in place in the hosts. Pageranks are incrementally computed as documents are added or deleted. A detailed discussion of the implementation of this algorithm is beyond the scope of this paper. On a data set of 1000 documents running on eight processors on a 100-megabit local area network, an average of 1846 messages were required to converge, with an average running time of 68.6 seconds.

6 Runtime Functionality for CoorSet

The requirements for experimental research on distributed coordination systems in CoorSet are: implementation of timed reliable asynchronous broadcast, configuration and realization of coordination systems on distributed resources and a runtime system which supports the extended associative interactions coordination model specified in Section 3. This section defines and describes the capabilities for these three requirements currently implemented for the CoorSet development environment.

A system for supporting experimental research on distributed systems needs the following capabilities:

• **Discovery of available hosts.** A distributed launching system must be able to find out what hosts are available to participate in the experiment.

• **Request authentication.** Any system of this type must ensure that only requests with the proper security credentials are honored.

• **Filesystem independence.** A distributed system should not assume a shared file system. Therefore it is its responsibility to see that binaries are transported to the execution sites when launching.

• **Host independence.** The system should handle the loading and execution of code on a variety of architectures.

• **Dynamic system structure.** Such a system should allow dynamic structuring of experiments, as these experiments will often involve joining and leaving protocols, and fault tolerance.

The CoorSet development environment uses the "Component Starting Component" (CSC) [14], an environment for launching Java components to configure and instantiate coordination systems for execution on distributed resources. The CSC is deployed on participating systems in a network. Once installed on a host in a network it can stay resident indefinitely. A coordination system is initialized by multicasting a solicitation for available hosts to discover CSCs without *a priori* knowledge of their locations. After receiving service offers, the program initializing the coordination system connects to an appropriate set of responding hosts, and then sends Java bytecode data and startup instructions. The CSC loads the component and starts the component in a different thread in its local Java Virtual Machine (JVM). Communication is guarded by cryptographic signatures to prevent unauthorized use. The CSC provides automated support for distributed systems research that does away with the necessity of manually logging into a number of remote workstations to launch the components of a system. The "main program" generated by the CoorSet compiler consists of a number of instructions to a network of participating CSC units to launch the components of the application across available hosts.

Once the CSC is installed on a network connected by timed asynchronous reliable broadcast then distributed coordination systems can be instantiated in minutes or even seconds. This allows a CoorSet program to be launched from a single point. The runtime system assigns a unique index to each component, which allows components of a like type to differentiate themselves. The runtime system assigns these in a predictable manner based on the format of the configuration file. This allows components running the same code to behave differently if so desired by choosing a control path based on that identifier. This identifier can also be used in interactions where a unique identifier is desired, such as for point-to-point communications, or identities in an election, just to name a few.

The Associative Interactions runtime listens to broadcast messages on the underlying network substrate, evaluates selectors, and forwards matching messages upwards to the application. Data flow semantics are now supported with a component of the run-time system that implements firing rules. Firing rule components inherit standard code that listens to messages and waits for the rule to be satisfied. The computational code connected to that rule is then executed.

Compound nodes inherit code that provides an event dispatcher for the accepts interface, and a standard application programming interface (API) for runtime modification of the profile, accepts and requests interfaces.

7 Implementation

The CoorSet language compiler, runtime system, and Component Starting Component are implemented in Java. CoorSet executes either with an implementation based on the Light-weight Reliable Multicast Protocol [11] that operates on top of IP Multicast, or Scribe [12], a multicast overlay that runs on top of the peer-to-peer Pastry [13] protocol. The latter implementation allows for implementation over wide area distributed systems. The Associative Interface, a class which mediates communication between the application and the network, listens on the multicast socket and evaluates the selectors of incoming broadcasts as the application invokes the message reception API. Matching messages are delivered, and the rest are discarded.

The CoorSet compiler generates Java classes for each defined component type, and a main program invoked to start the system. These generated classes use methods provided by the programmer for the computational part of the component as well as the CoorSet library.

The Component Starting Component is also written entirely in Java. When it receives components to launch, they are launched in independent threads in the same virtual machine. The Java Cryptographic Extensions (JCE), now a standard part of the Java SDK, provide the cryptographic primitives for secure key generation, signature generation, and signature verification for code bundles.

The broadcast model of communication allows greater efficiency of communication on systems like Ethernet where broadcast is the norm, requiring small numbers of messages to reach large numbers of recipients. In the data fitting example in section 5.2, a single message is all that is required to request processing by all available fitter units, instead of dispatching a separate message to each one. This represents a savings when some or all of the units are in the same broadcast domain, and invokes all units that are available, allowing for transparent replication of fitter units. The invoker can compare results for consistency to guard against faulty units, and choose the one that best satisfies a chosen metric amongst the various kinds of fits available.

8 Related Research

The previous paper [6] positioned the associative broadcast based coordination model in classification proposed by Papadopoulos and Arbab [15] and related it to other coordination models and languages. Since this paper focuses on a development environment for coordination systems in associative broadcast the related research is that which enables development of applications as coordination systems including languages, runtime systems and graphical specification environments.

8.1 Runtime Systems

Runtime implementations of coordination models and peer to peer systems are essential for experimental research on distributed implementations of coordination systems. The most closely related research to the runtime system described herein is Klava [16], which implements the Linda [5] model on top of an infrastructure that supports mobile code in a distributed tuple space, including a facility for transporting Java code across a network and starting it in a remote location.

Picco and Buschini [17] describe Linda in a Mobile Environment (LIME) that uses the Linda tuple-space model, dividing the tuple-space amongst a number of mobile agents. They extend the model by allowing the tuple space to contain classes, and using it as the code basis for a class loading mechanism, instead of the local disk.

SPACETUB [18] is a simulation environment for Linda-style languages, rather than an actual production environment. Each language is modeled in UML, and interpreted by the modeled class whose methods are the primitives of the language under consideration. Although intended to evaluate Linda-like languages, SPACETUB itself could be used as a coordination language by agents directly invoking SPACETUB's primitives.

Peer-to-peer networks can be viewed as a special case of a coordination model, where coordination is accomplished entirely on a set of agreed upon protocols for interaction. In this way each has a well-defined interface and a method for interacting with peers to request and provide services. The associative broadcast coordination model can be viewed as a peer to peer system with protocols for discovery of services and remote procedure calls. In peer-to-peer and associatively coordinated systems, connections are ephemeral, and exist only so long as two components are actually interacting. Although coordination models are commonly more structured than this, those particular models are part of a subclass of all coordination models. In general, a coordination model makes no such assumptions of communications medium, or the nature of connections between components.

8.2 Coordination Languages

HOBS [28], a higher-order calculus for broadcasting systems, models many of the important features of the bare Ethernet. It gives a calculus for reasoning about broadcast systems, including using "filters" on incoming broadcasts. This system has been implemented in the ML-descended language OCaml.

PiLar [19] is an architecture description language that uses the π-calculus to describe its semantics. Each component is also abstracted with a series of exported ports which can be connected to other ports in a strictly point-to-point fashion. PiLar was originally created to describe software architectures, but it is shown that it can be used as a coordination language, and an example of implementation of the Linda [5] model is given.

Manifold [20] is a language which collects groups of components into *manifolds* and encapsulates them into an independent process with its own virtual processor, having its own set of external ports and interconnections amongst the encapsulated members, and reactions to events and other changes in state of the members.

Components and manifolds are connected explicitly point-to-point, and once constructed, the system remains static.

Law-Governed Linda (LGL) [25] extends Linda by introducing a *controller* for each entity in the Linda network which mediates communication between the entity and the shared tuplespace. It enforces a set of rules called the *law* of the system on how entities read and write tuples. These rules are expressed in a predicate calculus such as Prolog.

Coordination Contracts [26] express relationships between objects in a business model. A *contract* between a number of *partners* represents an agreement that certain invariants will always be maintained, and that actions of partners will be coordinated with local actions. These actions are in the language itself, specifying what an object will do when a guard condition is satisfied.

CoML [21] is an XML-based language that describes the interconnections between a group of components implemented in a general-purpose language, for components that operate in an interconnection platform such as CORBA, JavaBeans, or .NET. It uses an event-driven model for communication, where there are event sources that fire events when conditions are met, such as during state changes, event sinks that react upon them, and event data. Components are composed by describing their interfaces and explicit connections to other components. Connections are changeable during runtime.

Linear Objects [29] are an integration of logic and object programming where the "facts" in a knowledge base include methods defined by classes. Program clauses may have multiple "heads" including references to these methods connected by an operator closely related to logical disjunction. Generation of a search tree creates references to the method signatures. Each step in generation of a search tree corresponds to a restricted form of broadcasting an associatively addressed message.

A CoorSet program is equivalent to a parallel production rule program [30] where both the rules and the object store are distributed. There are two object types: a rule object type has some member variables and three methods, a condition evaluation method, a conflict resolution method and an action execution method, a data object type has some member variables and two methods, an access method and a distribution method.

Web Services Flow Language (WSFL) [22] describes interactions amongst web services in either a flow model, which illustrates a particular business process, or a global model which describes how a set of web services interact without regard to a particular application, but this language is geared specifically towards web services specified in the Web Services Description Language (WSDL). Explicit connections are made between service instances, and using web service interfaces. Grid services [24] address the same idea as web services, but in the context of the computational grid. Services here also use WSDL, but extend it to allow stateful services, discovery, and use of the standard authorization mechanisms present in a grid.

WSFL's successor, Business Process Execution Language for Web Services (BPEL4WS) [23] describes relationships between business entities which use web services for all interaction. It also allows the separate specification of public protocols from private, internal protocols, further underscoring the need for components to be viewed as black boxes with a well-defined observable behavior, irrespective of how the internals work. This allows internal processes to be modified

as needed, while still maintaining the same public behavior and protocols. It abstracts web services-style interactions into "partner links."

9 Summary and Future Research

The CoorSet development environment appears to provide a capability for readily constructing applications in the extended associative interactions coordination model. The CSC provides a capability for easily constructing and executing experiments.

Future research will focus on formulation and evaluation of algorithms and applications in the CoorSet coordination model, development of an interactive interface for composition of CoorSet programs, and upon providing a more flexible and powerful security mechanism for the CSC.

Acknowledgements

This research was supported by the National Science Foundation under grant number 0103725, "Performance-Driven Adaptive Software Design and Control." We also wish to express our gratitude to the reviewers who suggested additional related work for our consideration, as well as future lines of research.

References

1. Dolev, D., Dwork, C., and Stockmeyer, L.: On the minimal synchronism needed for distributed consensus. *Journal of the ACM* (1987) 34(1):77-97.
2. Turek, J. and Shasha, D.: The Many Faces of Consensus in Distributed Systems. *Computer* (1992) 25(6):8-17.
3. Bayerdorffer, B.: Associative broadcast and the communication semantics of naming in concurrent systems. Ph.D. dissertation, Department of Computer Sciences, The University of Texas at Austin (1993).
4. Bayerdorffer, B.: Distributed computing with associative broadcast. *Proceedings of the Twenty-Eighth Hawaii International Conference on System Sciences* (1995).
5. Gelertner, D.: Generative communication in Linda. *ACM Trans. Prog. Lang. Sys.*, (1985) 7(1):80-112.
6. Browne, J. C., Kane, K. and Tian, H.: An associative broadcast based coordination model for distributed processes. *Proceedings of COORDINATION 2002,* LNCS 2315, Springer-Verlag (2002) 96-110.
7. Newton, P. and Browne, J. C.: The CODE 2.0 Graphical Parallel Programming Language. *Proceedings of the ACM International Conference on Supercomputing* (1992) 167-177.
8. Dolev, D. and Malki, D.: On distributed algorithms in a broadcast domain. *Proceedings of ICALP* (1993) 371-387.
9. Kane, K. "The CoorSet Interface Definition Language." *Preprint.*
10. Mittermeir, R. and Wurfl, L.: Greedy Reuse: Architectural Considerations for Extending the Reusability of Components." *Proceedings of SEKE'96, the Eighth International Conference on Software Engineering and Knowledge Engineering* (1996).

11. Liao, T.: Light-weight Reliable Multicast Protocol. INRIA Technical Report (1998), http://webcanal.inria.fr/lrmp/lrmp_paper.ps
12. Rowstron, A., Kermarrec, A-M., Castro, M., and Druschel, P.: SCRIBE: The design of a large-scale event notification infrastructure. *NGC2001*, UCL, London (2001).
13. Rowstron, A. and Druschel, P.: Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, Heidelberg, Germany (2001) 329-350.
14. Kane, K. and Browne, J.C.: The Component Starting Component: an environment for distributed systems and peer to peer research. Department of Computer Sciences Technical Report TR-03-42, University of Texas at Austin (2003).
15. Papadopoulos, G. A. and Arbab, F.: Coordination Models and Languages. *Advances in Computers*, v. 46, Academic Press, August 1998.
16. Bettini, L., De Nicola, R., and Pugliese, R.: Klava: a Java Framework for Distributed and Mobile Applications. *Software – Practice and Experience* (2002) 32:1365-1394.
17. Picco, G. and Buschini, M.: Exploiting Transiently Shared Tuple Spaces for Location Transparent Code Mobility. *Proceedings of COORDINATION 2002,* LNCS 2315, Springer-Verlag (2002) 258-271.
18. Tolksdorf, R. and Rojec-Goldmann, G.: The SPACETUB Models and Framework. *Proceedings of COORDINATION 2002,* LNCS 2315, Springer-Verlag (2002) 348-363.
19. Cuesta, C., de la Fuente, P., Barrio-Solórzano, M., and Beato, E.: Coordination in a Reflective Architecture Description Language. *Proceedings of COORDINATION 2002,* LNCS 2315, Springer-Verlag (2002) 141-148.
20. Arbab, F., Herman, I. and Spilling, P.: An overview of Manifold and its implementation. *Concurrency: Practice and Experience* (1993) 5(1):23-70.
21. Birngruber, D.: CoML: Yet Another, But Simple Component Composition Language. *Proceedings of Workshop on Composition Languages* (2001).
22. Leymann, F.: Web Services Flow Language (WSFL 1.0). http://www-3.ibm.com/software/solutions/webservices/pdf/WSFL.pdf
23. Thatte, S. (ed.): Specification: Business Process Execution Language for Web Services Version 1.1. http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/
24. The Globus Alliance: Open Grid Services Architecture. http://www.globus.org/ogsa/
25. Minsky, N. and Leichter, J.: Law-Governed Linda as a Coordination Model. *Object-Based Models and Languages for Concurrent Systems*, LNCS 924, Springer-Verlag (1995) 125-146.
26. Andrade, L. and Fiadeiro, J.: Interconnecting objects via contracts. *UML'99 -- Beyond the Standard*, R.France and B.Rumpe (eds), LNCS 1723, Springer-Verlag (1999) 566-583.
27. Sankaralingam, K., Sethumadhavan, S. and Browne, J.C.: Distributed Pageranks for P2P Systems" *Proceedings of the Twelfth IEEE International Symposium on High Performance Parallel and Distributed Systems* (2003) 58-69.
28. Ostrovský, K.: Higher Order Broadcasting Systems. Thesis for the Degree of Licentiate of Philosophy, Göteborg University (2002).
29. Andreoli, J-M. and Pareschi, R.: Linear Objects: Logical Processes with built-in Inheritance *Proceedings of 7th ICLP* (1990) 495-510.
30. Wu, S. Y., Miranker, D. P., and Browne, J. C.: Decomposition Abstraction in Parallel Rule Languages. *IEEE Transactions on Parallel and Distributed Systems* (1996) 7(11):1164-1184.
31. Page, L., Brin, S., Motwani, R., and Winograd, T.: The pagerank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project, 1998.
32. Chazan, D. and Miranker, W.: Chaotic relaxation. *Linear Algebra Applications*, 2 (1969) 199-222.