COMET: Coordination Middleware for Decentralized Distributed Environments*

Zhen Li and Manish Parashar
The Applied Software Systems Laboratory
Dept. of Electrical and Computer Engineering, Rutgers University, Piscataway, NJ, 08854, USA

Abstract

The scale, heterogeneity and dynamism of emerging distributed and decentralized environments make coordination a significant and challenging problem. In this paper we present COMET, a scalable peer-to-peer content-based coordination middleware infrastructure with flexible reactive behaviors and rich expressiveness. The COMET coordination model is based on a global virtual shared-space that can be associatively accessed by all peer nodes in the system and access is independent of the physical location of the tuples or identifiers of the host. Dynamically constructed, transient coordination spaces are also supported to enable context locality to be explicitly exploited for improving performance. The design, implementation and experimental evaluation of COMET are presented.

1 Introduction

The emergence of wide-area distributed and decentralized environments, such as pervasive information systems, peer-to-peer systems, and computational Grid infrastructures, has enabled a new generation of applications that are based on seamless access, aggregation and interactions. Examples include pervasive applications that leverage the pervasive information Grid to continuously manage, adapt, and optimize our living context, crisis management applications that use pervasive conventional and unconventional information for crisis prevention and response, medical applications that use in-vivo and in-vitro sensors and actuators for patient management, scientific and engineering simulations of complex physical phenomena that combine computations, experiments, observations, and real-time data to provide important insights into complex systems, and business applications that use anytime-anywhere information access to optimize profits. The defining characteristics of these emerging systems and applications are:

- **Heterogeneity**: The environments aggregate large numbers of independent and geographically distributed computational, communication and information resources. Similarly, applications typically combine multiple independent distributed software elements such as components, services and data sources.

- **Dynamism**: The computation, communication and information environment is continuously changing during the lifetime of an application, including the availability and state of resources and services. The applications similarly have dynamic runtime behaviors where that organization and interactions of the elements can change based on context, content and state.

- **Uncertainty**: This is caused by multiple factors including: (1) dynamism, which introduces unpredictable and changing behaviors that can only be detected and resolved at runtime, (2) failures, which have an increasing probability of occurrence as the system scales increase, and (3) incomplete knowledge of global system state, which is intrinsic to large decentralized and asynchronous distributed environments.

Together, these characteristics result in significant application development, configuration and management challenges that span all levels, including the programming model and system, runtime, middleware and operating system. Especially, enabling flexible and robust coordination becomes a significant and challenging problem in these environments.

Coordination can be defined as managing the runtime dependencies and interactions among the elements in the system. These dependencies and interactions can be complex and various (e.g. peer-to-peer, client-server, producer-consumer, collaborative, at-most/at-least/exactly, etc.), and both coordinated entities and the nature of the relationships and interactions between them can be ad hoc and opportunistic. As a result, realizing these coordination behaviors using low-level protocols becomes extremely difficult (if not impossible).

---

*The research presented in this paper is supported in part by the National Science Foundation via grants numbers ACI 9984357 (CAREERS), EIA 0103674 (NGS), EIA-0120934 (ITR), ANI-0335244 (NRT), CNS-0305495 (NGS).
Coordination infrastructures address this issue by providing coordination as a middleware service, and enabling coordination behaviors to be separated from computational behaviors and be separately expressed and implemented. The coordination service supports abstractions to deal with the issues of data communication and synchronization as well as process cooperation and competition. Key requirements for such a coordination middleware include a high-level abstraction with simple and expressive syntax and rich semantics, flexible mechanisms that can realize the various coordination patterns, and a high-performance, scalable and robust implementation that addresses the system challenges outlined above.

Clearly, designing and developing a coordination middleware that meets all these requirements are non-trivial. A key issue in the design and development of such a coordination middleware is the choice of the underlying coordination model. Models based on direct communication, such as Remote Procedure Call, imply a strict coupling in time, place and name between the interacting entities. This is not suitable for the large decentralized systems where maintaining common knowledge about the names/identifiers, addresses of an end-point as well as the syntax and semantics of the interfaces is infeasible. In contrast, models based on programmable shared-space provide temporal and spatial decoupling, associative data access mechanisms and flexible reactive behaviors, and can deal with the incomplete knowledge and system dynamism and heterogeneity. As a result, these models have been popularly adopted by coordination middlewares for supporting the coordination among system entities or software agents (e.g., TuCSoN [14] and MARS [3]), for dealing with the system physical dynamism and uncertainties (e.g., Lime [11] and PeerWare [6]), and for exchanging data between heterogeneous components using XML format tuples (e.g., XMLSpaces [18] and MARS-X [4]). While these systems (discussed in more detail in the following section) have successfully demonstrated the power and feasibility of shared-space based coordination middleware, distribution and scalability in decentralized environments remain a challenge.

In this paper we present the COMET, a scalable peer-to-peer coordination middleware infrastructure with flexible reactive behaviors and rich expressiveness. The COMET coordination model is based on a global virtual shared-space constructed from a globally known semantic multi-dimensional information space. The information space is defined by the ontology used by the coordinated entities, and is deterministically mapped, using a locality preserving mapping, to a dynamic set of peer nodes in the system. The resulting peer-to-peer information lookup system maintains content locality and guarantees that content-based information queries, using flexible content descriptors in the form of keywords, partial keywords and wildcards, are delivered with bounded costs. Using this substrate, the space can be associatively accessed by all peers in the system and access is independent of the physical location of the tuples or identifiers of the host. Dynamically constructed, transient coordination spaces are also supported to enable applications to explicitly exploit context locality. The current implementation of COMET builds on the JXTA peer-to-peer substrate and is deployed on a distributed network within Rutgers University. An experimental evaluation of COMET is presented.

The rest of this paper is structured as follows. Section 2 presents some background and discusses related work. Section 3 describes the COMET coordination model. Section 4 presents the architecture and implementation of COMET. Section 5 presents an experimental evaluation. Section 6 presents a conclusion and outlines current research directions.

2 Background and Related Work

2.1 Shared-space Based Coordination and Linda

The shared-space based model for coordination was made popular by Linda [5], which defines a centralized tuple space that provides a shared message repository to exploit generative communication [8]. The key attributes of Linda include: (i). Asynchronous communication that decouples senders and receivers in space and time. An inserted tuple will exist independently in the tuple space until it is explicitly removed by a receiver, and tuples are equally accessible to all receivers but bound to none. (ii). An associative multicast medium through which multiple receivers can read a tuple written by a single sender using a pattern-matching mechanism instead of the name and location of the producer. (iii). A small set of operators (write, read, and remove) providing a simple and uniform interface to the tuple space.

In Linda, a tuple is an ordered sequence of typed fields and a single tuple space is a multiset of tuples that can be accessed concurrently by several processes using simple primitives. Tuples are put to the tuple space by executing the out(t) operation, extracted using the destructive primitive in(t) and read using the non-destructive primitive rd(t), where t is the tuple and r is a template that matches the tuple. If multiple tuples match a template, one tuple will be nondeterministically returned. Both in and rd are blocking operations. The template r may contain wildcards that are matched against actual values in a tuple during the associative matching process. For example, a tuple ("task", 12) will be matched by the template ("task", ?Integer).

The rest of this paper is structured as follows. Section 2 presents some background and discusses related work. Section 3 describes the COMET coordination model. Section 4 presents the architecture and implementation of COMET. Section 5 presents an experimental evaluation. Section 6 presents a conclusion and outlines current research directions.

2 Background and Related Work

2.1 Shared-space Based Coordination and Linda

The shared-space based model for coordination was made popular by Linda [5], which defines a centralized tuple space that provides a shared message repository to exploit generative communication [8]. The key attributes of Linda include: (i). Asynchronous communication that decouples senders and receivers in space and time. An inserted tuple will exist independently in the tuple space until it is explicitly removed by a receiver, and tuples are equally accessible to all receivers but bound to none. (ii). An associative multicast medium through which multiple receivers can read a tuple written by a single sender using a pattern-matching mechanism instead of the name and location of the producer. (iii). A small set of operators (write, read, and remove) providing a simple and uniform interface to the tuple space.

In Linda, a tuple is an ordered sequence of typed fields and a single tuple space is a multiset of tuples that can be accessed concurrently by several processes using simple primitives. Tuples are put to the tuple space by executing the out(t) operation, extracted using the destructive primitive in(t) and read using the non-destructive primitive rd(t), where t is the tuple and r is a template that matches the tuple. If multiple tuples match a template, one tuple will be nondeterministically returned. Both in and rd are blocking operations. The template r may contain wildcards that are matched against actual values in a tuple during the associative matching process. For example, a tuple ("task", 12) will be matched by the template ("task", ?Integer).
2.2 Related Work

Several research projects and commercial products have successfully adopted and enhanced the Linda shared-space coordination model to construct robust coordination platforms. Examples include JavaSpaces (Sun 2000) [7], TSpaces (IBM 1998) [9], MARS [3], MARS-X [4], and XMLSpaces [18]. XML format tuples increase language expressiveness and support flexible tuple querying [18], e.g., DTD (Document Type Definition) and XML schema based searches. Most of the XML space implementations follow one of two approaches. Object-oriented systems (e.g., MARS-X [4]) closely relate tuples to objects (typically Java objects). Document-centric systems, e.g., XMLSpaces [18] treat tuples as XML documents that are stored as native XML.

These systems have also introduced programmable reactivities to enhance control and flexibility so that behaviors of the space can be tailored according to application requirements. For example, the primitive operations are mapped to logic events in TuCSON [14], which allows the space to react to the access operations. Similarly MARS [3] supports the definition of a meta-level tuple space and the association of reactions to access events. These models work well for the mobile agent coordination, Internet information retrieval, workflow management and E-Commerce. While these extensions to the original Linda model provide improved language expressiveness and control flexibility, they maintain the centralized tuple space implementations (possibly using partial replication) and the client-server architecture, which seriously limits the scalability of these systems.

More recent coordination systems have focused on fully decentralized architecture to address scalability [13] in widely distributed systems, for example, Lime [11], PeerWare [6], Tota [10], PeerSpace [2], etc. The two most related to this research, Lime and PeerWare, are described below. Lime [11] builds on the concept of Global Virtual Data Structures (GVDS). Its approach is to realize a global data space that is transiently shared and dynamically built upon the local data spaces of a set of hosts and the operations performed locally can have global effects. The primary objective of Lime is to provide Linda-like coordination in mobile environments. It exploits a flat data structure and extends the Linda interfaces with a location parameter \(\lambda\), expressed in terms of agent or host identifiers. PeerWare [6] also builds on the GVDS concept. It realizes a forest of trees, composed of nodes and documents, which are contributed by each peer. Its basic access primitive \(\text{execute}\) requires the parameters \(F_N\) and \(F_D\) functions to filter the operated set of documents and nodes. All the above system implicitly employ the context-aware programming style where information about the location of system components (e.g., nodes, hosts or agents) is required by the coordination primitives.

3 The COMET Coordination Model

The overall goal of COMET is to enable scalable peer-to-peer content-based coordination with flexible reactive behaviors and rich expressiveness, in large-scale decentralized distributed environments. The COMET coordination model is based on a global virtual shared-space, which is constructed from a globally known semantic multi-dimensional information space. The information space is defined by the ontology used by the coordinated entities. The space can be associatively accessed by all peers in the system and access is independent of the physical location of the tuples or identifiers of the host. Dynamically constructed, transient coordination spaces are also supported to enable context locality to be explicitly exploited for improving performance.

COMET builds on a Global Virtual Data Structure (a distributed hash table), similar to existing coordination middleware systems such as Lime and PeerWare. However these systems support a context-aware coordination approach, i.e., the identifiers of the destination peers are required as parameters in their interfaces. Maintaining such a global knowledge in large and highly dynamic distributed systems is not feasible. In contrast, COMET uses the context-transparent approach, where all operations use only content and are independent of the current state of the system and the mapping of content to these peer nodes.

The COMET model consists of layered abstractions prompted by a fundamental separation of communication and coordination concerns. These two abstraction layers are described below. Their implementations are discussed in the following section.

3.1 The Communication Abstraction

The COMET communication abstraction layer provides an associative communication service and guarantees that content-based information queries, specified using flexible content descriptors, are served with bounded costs. It supports content-based discovery, routing and messaging. This layer essentially maps the virtual information space in a deterministic way to the dynamic set of currently available peer nodes in the system, while maintaining content locality. It thus manages system scale, heterogeneity and dynamism.

The communication abstraction provides a single operator: deliver \(\mathcal{M}\). The message \(\mathcal{M}\) consists of (1) a semantic selector that is flexibly defined using keywords, partial keywords and wildcards from the information space, and specifies a region in this space, and (2) a payload consisting of the data and operation to be performed at the destination. This operator forwards the message \(\mathcal{M}\) to all destination nodes containing content that lies in the region defined by the semantic selection, i.e., that matches the selector. Note
that the resolution of this operator depends on the current information existing in this system. Further note that, unlike low-level messaging protocols that send/receive messages to/from specific destinations, the destination(s) in this case are dynamically determined based on the state of the system.

Coordination operations can now be directly built on the deliver operator. For example, the Linda-like operation \( \text{out}(t) \) can be implemented as deliver \( (\mathcal{M}) \), where the semantic selector is defined by the elements of the tuple and the payload includes the tuple and action “store”. The \( \text{rd}(\mathcal{T}) \) can be similarly implemented using the template \( \mathcal{T} \) to define the semantic selector and the payload including the action “read”. Note that the deliver can return one, all or some of the matched tuples.

### 3.2 The Coordination Abstraction

The coordination requirements of decentralized distributed systems require an extension of the traditional data-driven space-based coordination model. One such extension adapts concepts from event-based systems to support reactivity to events in the coordinate space. For example, the TuCSoN [14] and MARS [3] models support reactivity based on communication events. Similarly, PeerWare [6] supports reactivity to changes in system state and to data access operations.

The COMET coordination abstraction also extends the traditional data-driven model with event-based reactivity to changes in system state and to data access operations. It defines a reactive tuple abstraction, which consists of additional components: a condition that associates reaction to events, a guard that specifies how and when the reaction will be executed (e.g., immediately, once). The condition is evaluated on an access event. If it evaluates to true, the corresponding reaction is executed. Note that it is left to the application developer to ensure the logical correctness of the reactive behaviors.

The COMET coordination abstraction provides the following primitives:

- \( \text{Out}(\mathcal{T}, t) \): a non-blocking operation that inserts the tuple \( t \) into the space \( \mathcal{T} \).
- \( \text{In}(\mathcal{T}, \mathcal{T}) \): a blocking operation that removes a tuple \( t \) matching template \( \mathcal{T} \) from the space \( \mathcal{T} \) and returns it.
- \( \text{Rd}(\mathcal{T}, \mathcal{T}) \): a blocking operation that returns a tuple \( t \) matching template \( \mathcal{T} \) from the space \( \mathcal{T} \). The tuple is not removed from the space.

These basic operations operate on regular as well as reactive tuples and retain the Linda semantics, i.e., if multiple matching tuples are found, one of them is arbitrarily removed and returned with actual values assigned to the formal fields of that template. If there is no matching tuple, the operation waits for one to appear. Additionally, \( \text{RdAll} \) and \( \text{InAll} \) operators are defined to return all tuples that match a template.

### 3.3 Transient Spaces in COMET

Coordination middlewares based on the model outlined above are naturally suitable for context-transparent applications. Furthermore, since the underlying implementation maintains content locality in the information space, it is both scalable and flexible. However, certain applications, e.g., mobile applications, require context locality to be maintained in addition to content locality, i.e., they impose requirements for context-awareness. The uniform operators provided by COMET do not distinguish between local and remote components of a space. While this is a convenient abstraction, it does not maintain context locality and may have a detrimental effect on system efficiency for these applications. To address this issue, COMET defines transient spaces that have a specific scope definition (e.g., within the same geographical region or the same physical subnet). These spaces can be dynamically created. An application can switch between spaces at runtime and can simultaneously use multiple spaces.

The concept of transient spaces has been used in Lime [11] to enable sharing between hosts. In Lime, each host consists of multiple agents, and each agent can be equipped with multiple tuple spaces. Transient spaces can be established to enable sharing between the spaces belonging to agents co-located on a host or spaces on multiple connected hosts. The latter space is called a federated tuple space. The transient space in either case is a merge of the individual spaces. To guarantee the consistency of the federated tuple space, Lime defines engagement and disengagement protocols, which are executed as distributed transactions in the group when a host joins or leaves. Access to these spaces explicitly requires location information. In COMET, the spaces are always shared and transient spaces are used to enable applications to exploit context locality. The transient spaces have exactly the same structure and semantics as the original space.

### 4 Design and Implementation of the COMET Coordination Middleware

A schematic overview of the COMET system architecture is shown in Figure 1. The current prototype has been implemented on Project JXTA [15], a general-purpose peer-to-peer framework. The coordination space is provided as a JXTA peergroup service that can be concurrently exploited by multiple applications. The design and implementation of the COMET coordination and communication layers are described below.
4.1 Communication Layer

The communication layer of COMET is built on the Meteor messaging substrate [12], which provides scalable content-based routing and data delivery operations. Meteor consists of a structured self-organizing overlay and the Squid content-based routing engine.

Squid [16] provides a decentralized information discovery and associative messaging service. It uses a locality preserving and dimension reducing indexing scheme, based on the Hilbert Space Filling Curve (SFC) [1], to effectively map a multi-dimensional information space to the peer identifier space and to the current peer nodes in the system. The peer nodes form a structured overlay. The resulting peer-to-peer information system supports flexible content-based routing and complex queries containing partial keywords, wildcards, and ranges, and guarantees that all existing data elements that match a query will be found. Keywords can be common words or values of globally defined attributes, and are defined by applications. In the case of COMET, these keywords are part of the common ontology used by the coordinating entities. The keywords form the multi-dimensional information space, i.e., keyword tuples represent points in this space and the keywords are the coordinates. A keyword tuple in Squid is defined as a list of $d$ keywords, wildcards and/or ranges, where $d$ is the dimensionality of the keyword space. A keyword tuple only containing complete keywords is called simple, and a tuple containing partial keywords, wildcards and/or ranges is called complex.

Content-based routing in Squid is achieved as follows. SFCs are used to generate a 1-dimensional index space from the multi-dimensional keyword space. Further, using the SFC, a query consisting of a simple keyword tuple can be mapped to a point on the SFC. Similarly, any complex keyword tuple can be mapped to regions in the keyword space and to corresponding clusters (segments of the curve) in the SFC. The 1-dimensional index space generated from the entire information space is mapped onto the 1-dimensional identifier space used by the overlay network formed by the peer nodes.

As a result, using the SFC mapping any simple or complex keyword tuple can be located. Squid provides a simple abstraction to the layer above consisting of a single operation: \texttt{post(keyword tuple, data)}, where data is the message payload provided by the messaging layer above. The routing for simple and complex keyword tuples is illustrated in Figures 2 and 3 respectively.

Figure 2. Routing using a simple keyword tuple in Squid: (a) the simple keyword tuple (2, 1) is viewed as a point in a multi-dimensional space; (b) the keyword tuple is mapped to the index 7, using Hilbert SFC; (c) the data will be routed in the overlay (an overlay with 5 RP nodes and an identifier space from 0 to $2^8$-1) at RP node 13, the successor of the index 7.

Figure 3. Routing using a complex keyword tuple (2-3, 1-5): (a) the keyword tuple defines a rectangular region in the 2-dimensional keyword space consisting of 2 clusters (2 segments on the SFC curve); (b) the clusters (the solid part of the circle) correspond to destination RP nodes 13 and 32, which are routed to.

The Meteor content overlay is composed of peer nodes, which may be any node in the system (e.g., gateways, access points, message relay nodes, servers or end-user computers). The peer nodes can join or leave the network at any time. The overlay topology is based on standard structured overlays. The current implementation of Meteor uses the Chord [17] overlay network where peer nodes form a ring topology. Advantages of Chord include its guaranteed performance and logarithmic in number of messages. Every node in Chord is assigned a unique identifier and maintains a finger table for routing. The lookup algorithm in Chord
enables the efficient data routing with $O(\log N)$ cost [17], where $N$ is the number of nodes in the system. An example of a Chord overlay network with 5 nodes is shown in Figure 4. The Meteor overlay network layer provides a simple abstraction to the layers above, consisting of a single operation: \texttt{lookup(identifier)}. Given an identifier, this operation locates the node that is responsible for it, i.e., the node with an identifier that is the closest identifier greater than or equal to the queried identifier.

4.2 Coordination Layer

The coordination layer implements the coordination abstraction and primitives. Its main components include a data repository for storing, pending requests, and retrieving tuples, a flexible matching engine, and a message dispatcher that interfaces with the communication layer to convert the coordination primitives to messaging operations and vice versa. Tuples and templates are represented as simple XML strings as they provide small-sized flexible formats that are suitable for efficient information exchange in distributed heterogeneous environments. A tuple/template in COMET is an XML string, in which the first element is the tuple’s tag and is followed by an ordered list of elements containing the tuple’s fields. Each field has a name followed by its value. The field name may act as the type function (as in Linda), and the value must be actual data for a tuple and may contains a wildcard (“*”) for a template. Names and tags used in tuples are assumed to be part of an ontology that is globally known to all peers nodes in the system and may be application specific. This ontology forms a multi-dimensional information space, which is used by the communication layer for content-based routing and messaging.

The Java code below illustrates the creation, insertion and retrieval of a tuple in COMET. In this sample code, a tuple tagged “contact” has fields “name, phone” with values “Smith, 7324451000”. It can be retrieved using a relevant template. The last parameter of the \texttt{In} operation defines the matching relations, i.e., “exact”, which requires a matched tuple has exactly the same structure as the template.

The data repository uses different strategies to store regular tuples and reactive tuples in memory. Since reactive tuples can encapsulate executable codes, they are implemented as Java objects. Regular tuples are composed of XML strings and are uniquely stored as DOM level 2 objects [19]. The main advantage of using DOM for representing regular tuples is that it can support flexible matching relationships [18].

4.3 Tuple Distribution and Retrieval

The COMET \texttt{Out, Rd} and \texttt{In} operations are implemented using Squid routing. Using the tag and fields of a tuple, each tuple/template is associated with a sequence of keywords, which are then used to generate the \textit{keyword tuple} required by the Squid \texttt{post} operator. It is assumed that all peer nodes agree on the structure and dimension of the information space used to define the keyword tuples.

The tuple distribution and retrieval process using \texttt{Out} and \texttt{In/Rd} operators is illustrated in Figure 5 and 6 respectively. Tuple distribution consists of the following steps: (1) Keywords are extracted from the tuple and used to create the keys for the Squid \texttt{post} operation. The payload of the message consists of the tuple and the coordination operation. (2) Squid uses the SFC mapping to identify the indices corresponding to the keyword tuple and the corresponding peer id(s). (3) The overlay \texttt{lookup} operator is used to route to the appropriate peer nodes. This operator maps the logical peer identifier to a JxtaId and sends the tuple using the JXTA Resolver Protocol. The \texttt{Out} operator only returns after receiving the \textit{Resolver Query Response} from the destination to guarantee tuple delivery. In the case of \texttt{In} and \texttt{Rd} operations, the templates are routed in a similar manner. These two operations block until a matched tuple is returned by the destination in a peer-to-peer manner.

4.4 Implementing Transient Spaces

Transient spaces are provided by COMET to enable applications to exploit context locality to improve performance. These spaces are dynamically defined over a specified scope (e.g., geographical area, network addresses, etc.)
and enable secure coordination within the boundaries of this scope. Membership and authentication mechanisms are adopted to restrict information access to the transient spaces. The structure of the transient space however, is exactly the same as the global persistent tuple space. Note that the global persistent tuple space is accessible to all peer nodes and acts as the default coordination platform.

The implementation of dynamic transient spaces is based on JXTA peergroup. The advantages of the peergroup are: (1) it creates a secure environment, where only member peers can access and publish protected contents; (2) a peergroup does not necessarily reflect the underlying physical network structure; and (3) a peergroup service consists of a collection of the service instances running on multiple peers of the group. If any one peer fails, the collective peer group service is not affected and the service is still available from other peer members.

Figure 7 illustrates transient spaces in COMET. In the figure the global space includes five nodes and a transient space is constructed using two nodes, 5 and 10. The transient space interface provides operations for creation, switch and destroy. The creation of a transient space is initiated using a reactive tuple which represents a request to create a space for a specified scope. The condition part of the tuple defines the scope of the space. The creation process consists of coordination service initialization, in which a peergroup is created and instantiated in each involved peer, and finger table stabilization, in which a node joins the group. Peer nodes can belong to several tuple spaces and the switch operator enables an application to dynamically switch between coordination services associated with these spaces. To destroy the transient space, each peer node in the peergroup stops the services and deletes its local instance of space.

5 COMET Operation and Evaluation

5.1 Overall Operation

The overall operation of the COMET middleware consists of two phases: bootstrap and running. During the bootstrap phase peer nodes join the group and exchange messages with the rest of the group. During this phase, the joining peer attempts to discover an existing peer in the system and to construct its routing table. It sends discovery messages to the group. If the message is unanswered after a pre-defined time interval (in the order of seconds), the peer assumes that it is the first in the system. If a peer responds to the message, the joining peer queries this bootstrapping peer according to the join protocol of the underlying overlay (Chord in our prototype), and updates routing tables in the overlay to reflect the join.
The running phase consists of stabilization and user modes. In the stabilization mode, a peer node responds to queries issued by other peers in the system. The purpose of the stabilization mode is to ensure that routing tables are up to date, and to verify that other peer nodes in the system have not failed or left the system. In the user mode, each peer node interacts as part of the coordination space to provide coordination services.

5.2 Experimental Evaluation

COMET has been deployed in a distributed network of 64 Linux-based computers connected by 100 Mbps full-duplex switches. Each node has an Intel(R) Pentium-4 1.70GHz CPU with 512MB RAM and is running Linux 2.4.20-8 (kernel version). Each machine serves as a peer node in COMET overlay. The experiments include measuring the average round trip time for each of the coordination primitives provided by COMET. For an Out operation, the measured time corresponds to the time interval between when the tuple is posted into the space and when it is delivered to the destination, i.e., the time between post and PostResponse in Figure 5. For an In/Rd operation, the measured time is the time interval between when the template is posted into the space and when the matched tuple(s) is returned to the application assuming that a matching tuple exists in the space, i.e., the time between post and receiving the tuple in Figure 6. This time includes the duration of template routing, repository matching and returning the matched tuple. The measurements use the native clocks of the peer nodes. Note that in these experiments we used a single JXTA rendezvous peer.

5.2.1 Evaluation of the Out Operation

To evaluate Out operations, regular XML tuples were used, which consist of randomly generated strings with fixed length. The average size of a tuple was 110 bytes. No reactive tuples were used. Furthermore, network traffic was modelled as poisson arrival Out operations with inter-arrival mean time of 10, 100, 500 or 1000 milliseconds.

Figure 8 shows the performance of the Out operations with inter-arrival mean time of 10ms, 100ms, and 1000ms, and a system size of 16 and 48 peer nodes. The Y axis is the average round trip time. The figure shows that round trip time is fairly independent of the traffic inter-arrival time. The maximum average time of the 48 peer node system is approximately 47ms, which we believe is acceptable. In the figure, the average round trip time does increase with the system size. This increase is mainly due to the Chord overlay which has O(log N) cost [17] where N is the number of nodes in the system, and the latencies within JXTA.

The behavior of the Out operation was also studied for different system sizes. In this experiment the mean inter-
arrival time was fixed at 500 ms. Figure 9(a) shows that this system scales well with increased number of peer nodes. When the number of peer nodes increases 8 times, i.e., from 8 to 64, the average round trip time increases only about 1.6 times. This is due to the logarithmic complexity of the Chord routing algorithm. Figure 9(b) plots the actual round trip time for 16 and 64 peer nodes. The random spikes in this figure are due to lower level network contention in the experiment environment, which affects the operation of the JXTA resolver service. The periodic updates of the Chord routing table may also affect the operation round trip time. Note that the frequency and amplitude of these spikes are higher in a larger system.

5.2.2 Evaluation of the In/Rd Operation

To study the behavior of the In/Rd operation, two experiments were conducted. The first experiment evaluated the average time required for data retrieval and extraction using In and Rd operations with different system sizes. The operation latency was measured for 25 Rd operations and 100 In operations. In this experiment we assumed that the tuples were previously stored into the space by Out operations. In the second experiment, the average time required for each single operation was measured for different numbers of tuples, with fixed system size of 4 nodes. The lengths of the tuples are fixed at 110 bytes. The tuples were generated with random strings. The results are shown in Figure 10(b), in which In/Rd operations scale well with the number of nodes and is largely independent of the number of tuples in the system. The average round trip time for Rd/In operations is approximately 105ms for the scenario with 2000 to 12000 tuples.

5.2.3 Evaluation of the Transient Spaces

In this experiment we study the behavior of transient spaces. The creation and service initialization time for transient space is about 13 seconds for a space of 2 peer nodes, and 26 seconds for a space of 4 peer nodes. The finger table stabilization time is about 5 seconds for 2 peer nodes and 15 seconds for 4 peer nodes. A large portion of the latency is the time required to initialize a JXTA peergroup and start JXTA services. Note that initialization is a one-time cost and setting up the transient space can exploit context locality and improve system performance in terms of operation latencies. We are working on further analyzing and optimizing these behaviors.

6 Conclusion

In this paper we presented COMET, a scalable peer-to-peer content-based coordination middleware infrastructure with flexible reactive behaviors and rich expressiveness. COMET provides a global virtual shared-space that can be associatively accessed by all peer nodes in the system and access is independent of the physical location of the tuples or identifiers of the host. The virtual space builds on an associative messaging substrate and a distributed hash table where the index space is directly generated from the semantic information space used by the coordinating entities. Dynamically constructed, transient coordination spaces are also supported to enable context locality to be explicitly exploited for improving performance. The design, JXTA-based implementation and evaluation of COMET were presented. Initial experimental results demonstrate that both the scalability and performance of the system are promising. Additional evaluation and analysis of the system, including cross-layer issues are ongoing. Current efforts include controlling access to tuples using credentials and incorporating leasing mechanisms to manage the lifetime of tuples.
References


