Bissa-A Scalable and Distributed tuple Space

Charith D. Wickramarachchi, Pradeep R. Fernando, Udayanga S. Wickramasinghe, Dulanjanie Sumanasena
Prof. Gihan Dias, Dr. Srinath Perera, Dr. Sanjeeewa Weerawarana.
Department of Computer Science and Engineering
University of Moratuwa
Sri Lanka.

Abstract—The idea of tuple spaces is based on the white-board design pattern & made its first appearance in the late 1980s. Similar to its mathematical interpretation, we identify tuple as a set of values separated by some kind of delimiter. For an example we can represent the name, birth year & the home town of a student using a simple tuple <John, 1986, Colombo>.

In our work, we have implemented a tuple space that could span across a peer-to-peer network. The distributed hash table based implementation is highly scalable as well as fault tolerant, and provides an easy to program simple interface for distributed application programmers. The BISSA implementation provides a simple communication model for Distributed application developers and even can be used as a shared memory middle-ware for distributed applications.

1. Introduction

After the introduction of the tuple space[1] model there has been many research done on the domain of tuple spaces. One main branch of this research is the attempt to make a scalable model of tuple spaces. On the other hand research is going on to implement tuple spaces in different environments. MobiSpace[2] is an example of a research that has been done to implement a tuple space in JavaME environments. Our intention through this research is to combine these two paradigms together to build a more versatile tuple space model that can span across different platforms such as browsers, standalone Java applications, services, clouds, etc. while retaining the scalable features of a tuple space in a distributed environment.

The tuple space design provides a convenient programming model for communication between applications, compared to general message passing models. In Message passing models, parties that communicate with each other have to be alive at the same time. The advantage of tuple space communication is that the parties that communicate using the tuple space need not be connected at the same time to engage in the communication. Furthermore the processes can coexist in the space/network without knowing each others existence. These two factors sum up to making tuple space model time and space decoupled, a feature very few parallel/distributed processing architectures posses.

While conforming to these features of a generic tuple space model, we try to describe our implementation of BISSA in this paper, in two folds. One fold is a browser based tuple space implementation which provides a communication and coordination middle-ware for browser applications. It is intended to be used as a scalable solution for a black board pattern based communication between web browser applications such as for Web Gadgets operating inside a single page, gadgets located within tabs or even for browser instances located in different geographic locations. The Other fold is a peer to peer tuple space implementation which acts as middle-ware for facilitating unified decoupled communication between and within web applications, Java applications and services in a highly scalable manner.

The BISSA browser tuple space is based on Javascript and provides a Java script based API to access the Space. The API enables applications to act as a standalone local tuple space(residing within browser memory) or to seamlessly integrate with the distributed peer to peer tuple space outside it's local context. The BISSA browser space is also implemented as a shindig[3] feature so that it can be used as an inter-gadget communication infrastructure.

BISSA peer to peer space is a DHT based tuple space implementation for Java applications. It will provide a distributed shared memory abstraction for applications, hiding underlying communication complexities from the application developer. Tuples added to the system will be distributed among the nodes. Operations of the peer to peer tuples space are implemented to tolerate the ad-hoc nature of the peer to peer systems.

2. Related Work
2.1. The Linda Model

In mathematics, a tuple is an ordered set of elements which may or may not be of the same type. In computing, a shared memory space depicts a logically shared memory that is physically distributed. The Linda model, developed by David Gelernter and Nicholas Carriero at Yale University describes the concept of a "tuple space", that is a shared memory space that stores tuples. It demonstrates the concept of decoupled communication between processes. According to the Linda model, if two processes need to communicate with each other they do not send messages or share a variable. Instead, they create a new object, named a tuple and place it in a common space, rather like a white board, where an interested party may come by and extract or read the message. The model describes four basic primitives that enable the users of the space to operate on the tuples or the space itself; eval, out, in and rd[4]. The “out” operation is used to add a tuple to the space and the “in” operation is used to extract the tuple from the space. The “eval” operation is used by the initial process to generate the processes that would be using the space for its operations while the “rd” operation is used to read the tuple without removing it from the space. This initial model has been attractive mainly due to its simplicity, and due to the model’s other strong features of orthogonality, and the spatial-and temporal-decoupling of concurrent processes[5]. Since its initial idea the model has undergone many improvements or additions due to extensive research, with changes made to its primary operations, such as blocking and non blocking reads and writes being added to its primitives. The Linda model has been tried and tested in many languages including C, Fortran, postscript and Scheme as base languages.

2.2. DTuples

DTuples is a peer to peer tuple space implementation built on top of distributed hash tables and based on the Linda model[6]. It adopts a fail restore fault model to restore a crashed node's state. DTuples' persistent tuple space consists of two levels, the Common level and the Subject level[6]. Both these levels are subsets of the tuple space. The Common level is accessible by all nodes, while the subject level is accessible only to agents that are bound to it. This will by default include the agent that created the subject and any other agents that later bind to it. The lifetime of a subject depends on the lifetime of the agents bound to it and its expiration time. The expiration time is counted when all agents bound to it leave the space. If such an agent returns to the space within the expiration time it can retain the subject and the tuples held in it. The implementation uses four primitives, in, read, out and copy-collect, used to copy tuples between the spaces.

DTuples uses Freepastry as its underlying overlay network implementation and handles fault tolerance by replicating the tuples and making all primitives transactional, employing JOTM as its transaction manager. It matches its tuples using the name of the tuple as its first argument. This name is used to get the hash key. Therefore a tuples matching the same template have the same name and are stored in the same nodes[6].

2.3. FreePastry

Freepastry is a scalable, distributed routing overlay for wide-area peer-to-peer applications[7]. It is implemented in pure Java language and the source code is available under an open source license. The network of Freepastry nodes are called the pastry ring. Pastry assigns a 128 bits long unique node id to each pastry node in the ring. The first node to join a particular pastry ring starts or bootstrap a new ring & new nodes join the existing ring by referring to a current pastry node in their bootstrap process. Pastry is a DHT implementation. Hence, given a <MessageKey,message> pair, it routes the given message to the node which has the closest numerical id value to the message Key. Furthermore each pastry node maintains a list of nodes that are numerically closest to the particular node. This set is identified as the leaf[7] node set.

Pastry’s routing algorithm finds the numerically closest node id using prefix based discovery. It can route a message with a given id to its numerically closest node in under \((\log(n)_{2}b_{n},\text{ceiling})\) in normal operation (b is a configuration parameter with typical value 4).

There are few well known peer-to-peer applications that have been built using the pastry routing overlay. Examples[8] includes PAST: a peer-to-peer archival storage, SCRIBE: group communication/event notification and SQUIRREL: co-operative web caching.

2.4. Hierarchical tuple Spaces

One approach to a tuple space implementation is the Hierarchical Tree Structure[9]. Generally the nodes in Hierarchical tuple Spaces are categorised into two types, namely Execution nodes and the Memory Nodes. Execution nodes would be running distributed processes in them and would be connected to parent memory nodes (arranged in a tree like structure), which in turn would be contributing to the distributed shared memory where tuples are stored or replicated along.
The hierarchical tuple space employs a hybrid replication scheme[9] for availability/fault Tolerance and a data coherency protocol for concurrency control. Basic primitives supported by this kind of a tuple space implementations are OUT (writing tuples to space), READ(querying tuples from space) and IN(removing tuples from space). Whenever a primitive is executed on a process, the respective tuple would traverse up the node hierarchy, replicating that tuple until a match (or root node) is found. If successful, the node where correct match occurred would start propagating the matching tuple down the tree to the execution node who initiated the tuple query.(ie:- IN/READ).

Improved extensions to this hierarchical scheme can be seen with visibility scoped sub tree structures[9]. Here the replication is done up to a certain depth (not upto root) of the tree so that tuple space clusters are built having their own tuple space memory. As a result execution nodes will be limited to a specific portion of the whole distributed environment and nodes outside the respective visibility scope won’t be able to join in to the distributed application. Although Hierarchical tuple spaces could reduce coupling between memory distribution strategy and application programming certain limitations exist, including poor scalability, static inflexibility of the deployment model and a processing overhead and complexity involved in implementing visibility scoped tuple spaces.

2.5. Web Gadgets

Web gadgets are html /Javascript/CSS content that operate within a single entity inside a web based application. They are basically specified in an XML declaration conforming to a specification standard, hence enabling it to be reused among different web platforms/applications without much of a problem. Being considered as independent contexts (even if they reside in a single web page) web gadgets can be utilised in a wide variety of applications spanning from a simple widget or a reusable components to full blown web applications integrating multiple gadgets across Internet. The XML declarations of a gadget is processed by an context known as a gadget container. It is a gadget container’s responsibility to render gadget layouts and controls, process meta-data, user preferences etc. and deploy ‘features’[3] as specified by declarative XML syntax of a gadget.

2.6. Shindig

Apache Shindig[3] is an OpenSocial[10] compliant gadget Container, providing the infrastructure needed to host and access XML based gadgets. Shindig comprises of a Javascript based gadget container which provides core gadget functionalities such as gadget communication, layout, security etc., a gadget Rendering Server which outputs XML based gadgets into Javascript and HTML and an OpenSocial container and Data server for OpenSocial based gadget communication.

Shindig provides different capabilities to gadgets it renders, through a mechanism called “Shindig Features”. Shindig features are Javascript libraries with some useful functionality provided either to gadgets, containers or both at the deployment time. Currently supported features[3] by Shindig vary from UI functionalities and content types support(ie:-Flash) to supporting OpenSocial API’s. Gadgets that need to use a specific feature should indicate them within their respective gadget headers.

Shindig Pubsub, a feature intended to facilitate inter gadget communication, provides a unified infrastructure through the publish subscribe messaging paradigm for XML based gadgets. This is similar to what we are going to achieve through BISSA browser tuple Space based communication. Shindig primarily supports three primitives, publish(), subscribe() and unsubscribe(). As mentioned before analogous to publishers and subscribers paradigm, “Shindig Pubsub” associates gadgets through a set of keys or channels, which would be used in publishing and subscribing to information that would be communicated between gadgets, effectively acting as an inter-gadget communication medium.

Whenever a gadget publishes information with in a specific event, all the subscribed gadgets relevant to that specific channel are notified and the event information are passed down. Gadgets can also unsubscribe at any point in time so that they are cut-off from information coming through the respective channel.

2.7. Tuple Space in Mobile Environments

After the tuple Space concept was introduced the implementation of tuple Spaces in different environments has been a popular area of research. Work has been done on implementing tuple Spaces on Mobile Environments. Project LIME (Linda in a Mobile Environments)[11] focuses on implementing a Linda-like tuple Space in mobile environments. It handles the complexities that comes with the resource constrains and the ad-hoc nature in the mobile environments and provides a simple interface to the programmers.

MobiSpace[2] is a middleware infrastructure that provides a distributed tuple space abstraction for Java Mobile Edition environments. Mobile programmers will be able to use this model to achieve loosely coupled communication using a standard programming model.
MobiSpace provides a primary-based replication where all the mobile nodes will be connect to a central computer where tuples are stored. And mobile nodes will be acting as secondary servers and they will be storing a selected portion of the space in the local space. The communication happens using the GPRS service, or any other Internet service available for mobile. MobiSpace also supports replication among the peers where mobile devices have formed a peer to peer network.

2.8. PAST

PAST is a large scale peer to peer persistent storage utility[8] that uses pastry as its routing overlay. Each stored file in PAST has a unique id attached to it. Applications can store a file within the peer-to-peer network by giving an id & the file to be stored. Upon receiving the file, PAST then routes the file as a pastry message. Once the file received by the appropriate pastry/PAST node it gets saved & replicated among the pastry leaf-set of that particular node. PAST can withstand high churn due its effective replication management system. PAST also supports the caching functionality for most recently used files. Each PAST node will have a uniform number of files if the id distribution of the files are uniform, thus ensuring dynamic content balance over the peer-to-peer storage.

Further descriptions & evaluations of PAST can be found in[8][12]

3. The BISSA Peer to Peer Space

3.1. Approach

BISSA global space is a distributed tuple Space and is based on peer to peer paradigm. It uses Freepastry[7], a distributed hash table based routing overlay to route the content based on their hash values. The space is created by the nodes which get connected to the peer to peer network. Application will get the abstraction of local tuple space, while in reality the tuples get stored in a distributed manner within the P2P network. In our implementation we used PAST [8] an archival tuple storage facility and extended it for our specific needs.

3.2. Primitives Supported

3.2.1. Write Operation

BISSA offers an operation "write" to insert a tuple in to the space. Underlying actions such as distributing and replicating data will be invisible to the user.

As an overview we can describe the write operation in two phases,
1. Write tuple as a DHT element with an associated hash calculated for it.
2. Update the Index files that are distributed in the system.

In the first phase we generate a 160-bit unique hash for the tuple and store it in the DHT. We use PAST[12], A storage system built on top of freepastry for storing tuples. Hence the tuple is stored in a node in the system which has the node id closest to the tuple hash.

The second phase is to update the tuple Indexes. We use an indexing mechanism to support the read operation of tuples.

As shown in the above diagram after a tuple is inserted to the space we update the indexes that represents it and matches the possible wild card queries for that tuple.

Ex : a tuple : <hello,world> can we quarried from a two tuple templates <hello,?> or <?,world>

In our system a wild card is represented by a null value. So a template tuple for <hello,*> will be <hello,null>.

The tuple indexes are actually list-like data structures that contain unique hash ids for a given template and a list mapping the template ids to the hash ids of actual tuples and the nodes where they are stored.

We update these indexes at the node where indexes are actually stored. The reason for that is, if we take the data structure to the local node that called the write operation and update it there and put it back to the remote node it can cause inconsistencies. Because in a distributed environment where a parallel updates are possible for a same index it can cause index to not having all ids of tuples that it must point.

So we use a message passing mechanism to do the updates at remote nodes.
To improve the performance we allow parallel updates with the index storage system with a node as well. But to achieve mutual exclusion and thereby avoid problematic scenarios as described above we use a locking mechanism based on a hash id, proved by a similar implementation in PAST. It provides the facility to access the storage system parallel but it only permits update on a same index hash id one at a time. That guarantees the fact that there can be parallel updates on different indexes in a same node but it does not allow to update the same index in parallel.

Using this implementation we achieved the mutual exclusion property on parallel updates while not unnecessarily constricting the access.

In our current implementation we support primitive types and Java object as tuple elements. Also we provide a way to user to specify non wild card elements with the tuple insertion so that space that will be used for indexing will be saved and also the Message traffic will be less, improving the system performance.

3.2.2. Read Operation

The read operation returns the tuples that match the template given as a parameter.

Example: <hello,world> and <hello,tuple> will be returned if user gives the template <hello,null> (given that <hello,world> and <hello,tuple> are in the Space)

Operation of the read operation is the simplest one. In that, first the system will generate the hash for the tuple template. And then it will lookup for the index structure associated with it. So that index will have all hash ids for the real tuples matches the template. Then System will get the real tuples using that hash ids and return to the user.

In our implementation we support two versions of the read operation,

1. A blocking read where users call the read is blocked till all the tuples are received.
2. Non blocking Asynchronous read there tuples received will be given to the user at the time they received. So that users will be not waiting till all the results available. They can register a call back handler with the operation and results will be given to that handler.

3.2.3. Take Operation

The operation differs from the read operation, that it remove all the tuples that match the given template. Removing tuple instance that has been replicated among the nodes of a peer-to-peer system raise some serious questions.

The BISSA take operation can be divided in to three phases,

1. The retrieval of the tuple from the peer-to-peer system that matches the given template.
2. Deletion of the tuple content from the peer-to-peer system.
3. Updating the indexes that carry wild card info of the deleted tuple.

We use a messaging based mechanism to delete the stored tuples right in their local nodes & avoid raise conditions through hash id based locking mechanism as with write operation. Still the BISSA tuple Space implementation does not guarantee the atomic execution of the take operation. It is possible to implement 100% consistent take operation with some agreement protocol, But still we prefer lightweight behaviour of the system as our goal is to implement highly scalable tuple space implementation.

The non-atomic behaviour does not break the consistency of the space at any time. For an example if the take operation failed during the index updating phase then there fill be false indexes pointing to a non-existing tuple. When read operation encounters a invalid index, it ultimately does not return the indexed tuple since it is not stored within the peer-to-peer storage.

The other major obstacle for removing all the instances of a particular tuple is the caching feature. When a node retrieves a tuple it gets stored in their local caches. The next time when there is a query for that exact tuple it retrieves the tuple from the local cache. We have provided a time stamped based caching implementation for the users who wish to make use of the cache. Users can specify the time-out of the cache, so that user don't mind getting a cached version of the tuple out from their local caches given that the threshold time has not expired(even if the tuple has been removed by the take operation).

3.3. Tuple Subscription Mechanism

Bissa provides a subscription mechanism for applications so that they can subscribe for tuple templates. Subscription can be done by giving a tuple template and a reference to a tuple Listener implementation.

After subscribing for a tuple, application will receive tuples that matches with the template when tuples are inserted to the tuple space.

There are some problems to be solved when implementing this mechanism.

1. How to distribute subscriptions
2. How to maintain the subscriptions with the ad-hoc behaviour of the space.
3. How the un-subscribing will work

One easy approach to implement this mechanism is when a application subscribe for a template the application can poll the tuple space time to time and determine the new tuples that were added to the space and give it to the application. But the problem with that approach is that will
cause lot of unnecessary message passing in the system and degrade the performance of the overall system. As a solution for that we use a data structure that is associated with the tuple indexes to keep subscriptions. So subscription data for a tuple template will be associated with the index that is for that template. When a tuple inserted and when that cause a update on that index (i.e. tuple must match the template of index). When that happens system will use the subscription data structures associated with index and notify the subscribers for that template. That will reduce the number of messages passed compared to the polling method.

Maintaining the subscriptions with the ad-hoc nature of the system is another problem to be solve. With the ad-hoc nature it can cause two major problems.

1. Nodes that keep the subscription data may be unavailable or dead so that can cause inconsistencies (mainly data losses).

2. Routine for unsubscribing. That will cause system to do unnecessary work since it will be sending unnecessary notification messages to nodes that are not there.

To avoid the problem of losing subscription data with the dead nodes we replicate both indexes and subscription data. That will give some level of fault tolerance for subscription mechanism and other operations.

Identifying dead nodes can be achieved by running a messaging protocol time to time. But as same as the polling method it will be costly in terms of system performance. As an alternative, without polling we do a checking for liveliness of a node only when it is needed to be notified. Till then its subscription will remain even though it has died. If liveliness check it returns that it is not alive system will remove the subscription data for that particular node. Still yet there can be a trivial scenario that will be not identified by above method which is if a node goes dead and another joins with the same node id and in that dying and joining time period no liveliness check was done. So system will not know that its a new node and subscription will remain. So as a solution when a node joins it send a join message to other nodes so that nodes that have subscription data can use this detail to update its data correctly.

4. BISSA Gadget Communication

4.1. Approach

BISSA browser Tuple Space resides in a local browser instance and integrates into the global Tuple Space. However depending on application programmers’ choice BISSA browser space can remain as an independent tuple space residing only in browser or as a fully integrated tuple Space that is being synchronised with dynamic tuple Space environment as well. Local tuple Space API is fairly consistent with the global tuple space, hence all the primitives available in global space are true for local space as well.

Design approach of local tuple space is primarily concentrated on managing tuples that are being queried and updated on the local space and synchronising local tuples corresponding to the global space. The core functionality is implemented in a Javascript library compatible in running under latest browsers. As in the case of global Tuple Space, a tuple of the form \{a1,a2,a3,....,an\} is the granularity of message passing process in local browser based tuple Space.

Local tuple Space can be considered as a combination of a tuple pool (TP) that keeps track of tuples in local space and a hash table instance (TPHT- tuples to processes hash Table) that associates tuples with the respective local processes who involve in the tuple exchange. Whenever a tuple is written into the local space, the tuple pool is updated accordingly and relevant processes are notified of the availability of tuples using the hash table. If a local process wants to query a tuple T or a specific template Te (a tuple with wild card entries in it), Local tuple Space Manager will match T or Te with the tuples set \{T1,T2,....,Tn\} in TP.

A special case occurs when matching tuple set with a given template Te. First matching algorithm compares the no of entries with each tuple in the set. If this is successful on a given tuple Ti in the set then tuple entries contained within Ti are individually matched bypassing any wild card that is encountered. If a mismatch occurs at any given entry, matching on Ti is considered unsuccessful. Ti is considered successful if an only if all the entries are matched with entries of Te.

4.2. Integration

BISSA provides access to global tuple Space for users who work within local browser instances. The main idea is to provide a uniform Interface that can seamlessly communicate with both local and global tuple Space memory easily and flexibly. Integration approach for BISSA infrastructure for local browser clients is through Web Services that wrap BISSA runtime instances. Prospective clients who wish to query tuples distributed in the global space (i.e.: in different geographic localities) can use the JavaScript API provided by local tuple Space runtime inside the browser instance. This mechanism also provides an efficient way of communicating between two or more browsers located in two different machines or even within two browser windows in the same machine.
BISSA local tuple Space connects to a BISSA web service node via a Javascript Stub that wraps an Ajax connectivity layer. The browser tuple Space will try to synchronize its localised tuple content during each connection session as required effectively appearing to be a tuple cache for local browser processes. As a result any process that queries for a certain tuple or a tuple template will be delivered all the conforming tuple content from the local tuple space as well as global Space. This model is pull based since browser tuple Space itself is responsible for pulling tuple content/data out from the globally scalable tuple space. The following diagram describes this overall model of integration

4.3. Messaging Layer

Local space queries for tuple content provided by the API are differentiated into two categories, synchronous and asynchronous. Synchronous method calls would wait until global Web service enabled global node replies for the tuple query while asynchronous calls would notify the node and resume execution. Users have the choice of adapting any of these variants depending on the requirement or performance considerations.

Sometimes it may be important that the browser application relieves the connection burden on global tuple Space and resume the local work it has been engaged in to increase performance of the application. BISSA asynchronous connectivity support this requirement by temporary delegating tuple queries to global layer and letting it do the work for you. The local tuple content will only be updated when these queries become successful reducing overall memory footprint on the browser runtime.

On the other hand, sometimes applications may want to constantly query content in a specific template. In this context handling and management of tuple queries would be pretty cumbersome to the application developer. Application programs can use subscribe option in these scenarios, to bind into global space for specific tuple queries. BISSA will take care of the periodic querying on global space for the respective tuples/templates and notifying the relevant processes which subscribed/bind to them.

4.4. Web Gadget Communication

With all these aforementioned features at our disposal, BISSA has the potential to become an extremely efficient communication infrastructure for web based applications and scripting based platforms. Usually web application integration spanning across the internet tend to be ugly and highly complex and coupled with the increase of number of integration units. This inherent nature of web applications can be exploited by BISSA to provide an ideal way of inter-communication, since this is exactly and fundamentally what BISSA tries to resolve, “providing a shared memory abstraction for naturally tightly coupled set of processes/entities”. So in-effect communication between these set of application entities/components can be facilitated through writing and retrieving tuples through BISSA (shared) tuple Space, greatly reducing complexity or coupling between the components involved.

In implementation aspects this concept of using BISSA as an inter-communication framework has presented us with several challenges. That is, where/in which aspect of web we should be implementing such a system? and how exactly it should be implemented to achieve an almost unified way of inter-communication? The answer lies in Web Gadgets. As stated previously on this paper, Gadgets are web content that operate within a single entity inside a web based application. Specified in xml syntax they are considered independent entities even when they are contained within a single web page. So as per the answer to our first question web gadgets seems to be a very good choice due to their independent and highly coupled nature. For example BISSA can present a shared channel for gadgets operating inside a web page or set of gadgets integrated across a network/internet. That in-effect is an efficient communication mechanism since otherwise complex message passing mechanisms could have deteriorated the overall system performance.

Web Gadgets also seem to be the ideal answer to our second question as well. The solution lies in “features” which are a standard compliant way to implement specific services for web gadgets. This is in the sense that, BISSA browser/local space runtime can be wrapped into a gadget feature, to be deployed on gadget servers at runtime. Therefore gadgets can be deployed in different gadget servers (provided we have required BISSA feature implementation for each different gadget server ie: Shindig, iGoogle, etc) exposing them to specific BISSA
runtime libraries creating an unified infrastructure for inter-gadget communication. Using this kind of BISSA enabled feature, gadget developers could use the functionality described above to facilitate communication between gadgets located in a single browser window (i.e., using local space API) or even between different gadgets located in separate servers or domains.

Following is high-level overview of how a BISSA enabled feature can be deployed on a Gadget Server. Note that this feature requires BISSA local Tuple Space runtime to be present in both container and gadget level at the deployment time. This is because gadgets and their containers are typically decoupled in gadget servers and gadgets are accessing local Tuple Space instance shared between them. BISSA feature is using RPC feature internally to delegate API calls between container and gadget level.

4.5. The Big Picture
If we predominantly focus on the big picture, what BISSA tries to create is an infrastructure that expands over intra-nets, possibly over the Internet on different platforms where each node is contributing to the distributed memory of the tuple Space. With local and global space integration in place we can see some of these nodes exposed as BISSA Web Services enabling independent browser instances to act as clients providing them the ability to access global space as well as their own local one's. This is shown in Fig 4.

4.6. Functions Supported
BISSA local space provides a clean and consistent API to support inter-gadget communication. Although related to generic tuple space API, functions supported makes into two categories.

a) Local tuple space API
b) Global tuple Space Integrated API

This differentiation gives application developers a clear separation of concerns whether to use a standalone local tuple space or to get exposed to the distributed tuple Space.

4.6.1. Local tuple space API
Following are the supported local space primitives.

a)bissa.read(t_template, callBack, subscribe)
read a tuple from the space relevant to the given template.

*callBack = the call back function that the requested tuple will be delivered when available in space or immediately, if already existing in space.

*subscribe = if true the user will be notified every time a tuple with the given template is added, else notified only when tuple is available in space OR immediately, if tuple is already in space.

_t_template can be either a tuple template (i.e., bissa.Tuple("a","??","??")) or a pure tuple (i.e., bissa.Tuple("a","b","c"))
b)bissa.take(t_template,callbck,subscribe)
remove the tuple from the Space and read it

callBack = the call back function that the requested tuple will be delivered when available in space or immediately, if already existing in space.

subscribe = if true the user will be notified every time a tuple with the given template is added and respective tuple will be deleted from space else notified only when tuple is available in space OR immediately, if tuple is already in space and will be deleted from space after that .

c)bissa.put(tuple)
insert a tuple into space , subscribed users will be notified if requested tuple is being inserted.

4.6.2 Global tuple space API
Following are the Global Space primitives supported,

a)bissa.read_global
read tuples to a given tuple/template from the tuple space . Tuples would be read from both global and local space. This API supports both asynchronous and synchronous messaging . Synchronous read requests to the tuple space would block until a available tuple is fetched from the global space. Asynchronous reads would immediately switch control to local space after dispatching the respective request to the global space . tuple read requests can be bind/subscribed to a template . tuple space will take care of fetching tuples from global space from periodic intervals for a specific timeout period. User can configure these parameters to suite their application requirement.

b)bissa.take_global
removes tuples to a given tuple/template from the tuple space . Tuples would be removed from both global and local space. Similar to the #global_read this API supports both asynchronous and synchronous messaging .

c)bissa.put_global(tuple)
insert a tuple into both global space as well as local space.

5. Results and Analysis

After successfully implementing the tuple space service we performed a scalability test and a latency test to measure the behaviour and performance of the peer to peer network while scaling up the system. The results and analysis of these test are as follows.

5.1. Scalability Test

The application we used for the test was a Monte Carlo simulation[13] for stock value changes for a company. This is an embarrassingly parallel algorithm, which involves independently executing disconnected components[14]. We used BISSA as a middleware to share tasks between processors and finally to aggregate the results; using a master/slave strategy where at start master will put tasks to the shared space and then after a starting command by the master, workers will take tasks from the space in parallel, execute them and submit the finish work to the space.

We used a control lab environment with machines with same configuration and did the scalability test with up to 35 computers.

5.1.1. Results Obtained

Figure 5 shows the Results we obtained from the scale test with the increasing number of machines.

![Figure 5: Speed up gained with the increasing number of nodes.](image)

5.1.2. Analysis

As we see in the results we were able to get an almost ideal parallelism with the scaling of the system. But since we tested only with 35 machines we can’t claim that we have a massively parallel implementation. But we can say with confidence that our system is a reasonably scalable one. The
results show a slight performance lag in the latter part of the results. This is due to the number of work per computer became less so as a result the communication latency came into effect on the results.

5.2. Latency Test
One other test performed on the BISSA peer to peer space was the Latency test. A latency test is used to measure time taken for an operation to complete. The idea of this test was to measure the time taken for the main BISSA operations; write, take and read to complete and analyze the effect of increasing number of nodes to its performance.

5.2.1. Results Obtained
Figure 6 shows the results obtained in measuring the time taken to perform one read and one write on an increasing number of processes.

![Figure 6: Operation latency with increasing number of nodes in the network.](image)

5.2.2. Analysis
According to figure 6 it is apparent that the latency of read and put operations are hardly affected by the increase in number of nodes connected to the space. This shows that in an environment where the message complexity is not so high, the performance is unaffected by the number of processes involved in reading and writing tuples to the space. Though these tests can hardly prove the system robust for a massively distributed system it did give satisfactory performance for the range of tests we have done, showing that the system would definitely give reasonable performance for some number of machines.

According to research conducted by Kato and Kamia[15] FreePastry behaves reasonably well up to about 800 nodes. Hence we can reasonably conclude that, BISSA being built on top of FreePastry and having showed good results to the tests we have conducted, that it would continue to perform well for an even larger number of nodes.

6. Applications

6.1. Role of Web Browser In Distributed Processing.

The world is moving from standard desktop applications to web-based applications. The modern operating systems such as Chrome OS establish this concept further. With browser becomes more prominent in the computing market, we should explore the possibilities of getting the processing power of the browser in to distributed processing environments.

Such applications can make use of BISSA as their underlying communication middleware. The peer to peer global tuple space will act as the shared memory between all the browser instances. JavaScript processing clients can fetch data from the space, process them & write back to the global space.

6.2. BISSA for SETI @ Home Like Applications

SETI at home like distributed computing efforts lacks a user friendliness to some extent since users need to download a client and contribute to the grid. But there can be considerable about of people who are willing to contribute and can't contribute since it requires them to download an application to the local computer and run it.

One solution we are purposing is to use the computation power given by the web browser. If we take modern web browsers it facilitate execution of Java scripts. System can give a URL to a user and it can go to web application that can run a Java script base client (ex : web Gadget ) so that gadget will get the jobs from a job pool and calculate and send back the results. So it will be a better solution than telling people to download a application. BISSA can be used to implement this kind of infrastructure easily since it provide a unified communication mechanism for Java applications and Java script gadgets.

6.3. Simple Weather App

The Weather Monitor is yet another example of possible applications of BISSA. It utilises the distribution of BISSA to gather and analyse weather data. Each node in the weather monitor may act as a weather collector, weather monitor, or
both. Weather collectors add tuples with weather information along with its location to the space. For example, a weather collector collecting temperature information at location id x and tuple id y will be updating temperature t with tuples of the form \(<x,y,temp,t>\). The weather monitor we implemented for demonstration purposes uses only temperature information, therefore its tuples take the form \(<x,y,t>\). The weather monitors query for matching tuples in constant intervals and updates its weather information accordingly. This querying is done using individual threads that handle new information coming in from each location.

6.3.1. Implementation

The implementation of the weather collector simply involves reading sensors periodically and putting the respective tuples. The Location ID indicates the location where the information has been updated and the tuple ID helps the monitor to determine if the tuple is the latest update yet. Each new temperature update from the same location is given a new tuple ID. The monitor updates its temperature information using separate threads to handle each location. Each thread performs periodical read() operations on the its location and if a tuple with a newer tuple ID is received its current weather information is updated. This information is then used to graphically present weather information on a map. The older tuples are periodically collected by the Weather collectors to keep the space from being overloaded by expired tuples.

Using BISSA to implement this application is beneficial in many ways. Being a distributed application that requires broadcasting information a shared memory space implementation proved best to serve its purpose. Also using a tuple space implementation enables the easy representation of data, making each update lightweight and easily accessible.

7. Future improvements

7.1. Extending BISSA to Mobile Spaces

As described above in our current implementation we have a two level tuple space implementation where one is a peer to peer tuple space implementation and other is a in browser tuple space implementation which act as a client to the peer to peer space.

As a future improvement to the our model we can extend this model to have mobile nodes as explained in MobiSpace[2] So they will be acting as a client space to the peer 2 peer space. Having this kind of Mobile Space integration with BISSA will be useful since. It will introduce a unified easy to program communication infrastructure for Java Applications , Web Gadgets and Mobile Applications.

8. Conclusion

Here we have presented the design & implementation details of BISSA, a middle-ware infrastructure based on tuple spaces for distributed and web based applications . We have shown how BISSA can comprehensively replace distributed but highly coupled message passing systems with a loosely coupled shared memory abstraction using an effective tuple Space communication paradigm.

We have also portrayed how BISSA has addressed the problem of mapping direct to tuple based content addressing by utilising a p2p based Distributed Hash-table. Furthermore this paper presented how BISSA browser tuple Space can act as a local tuple cache and shared browser instance to facilitate rapid web application integration and gadget communication.

On another perspective, BISSA can be considered as a light weight distributed application platform with a some what relaxed consistency model. This makes BISSA a very powerful and scalable middle-ware for a wide variety of web and Java based applications , acting as a unified communication channel to make different environments integrate easily and effectively.

BISSA is currently an active project with a working code base, which embeds different aspects we have portrayed throughout this paper. The website www.bissa.sourceforge.net offers updated information about the project and downloadable resources including BISSA source code.

REFERENCES


[8] A. Rowstron and P. Druschel, "Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility", ACM Symposium on Operating Systems Principles (SOSP'01), Banff, Canada, October 2001.


