2006

Tuple board: a new distributed computing paradigm for mobile ad hoc networks

Chaithanya Bondada

Tuple Board

A New Distributed Computing Paradigm for Mobile Ad Hoc Networks

By Chaithanya Bondada

A project report submitted in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

Rochester Institute of Technology

2003

Committee:

Prof. Alan Kaminsky, Chair

Prof. Hans-Peter Bischof, Reader

Prof. Michael Van Wie, Observer

Revision History

Version		Description	Author
No.	**Date**		
1.0	10/25/03	First Draft	Chaithanya Bondada
2.0	1/1/2004	Second Draft	Chaithanya Bondada
2.1	1/16/2004	Final	Chaithanya Bondada

Table of Contents

1 Introduction

The ***Anhinga Project*** at the Department of Computer Science at the Rochester Institute of Technology is developing a computing infrastructure that supports distributed applications on ad-hoc networks of small proximal wireless devices. Existing middleware infrastructure for distributed applications relies on central servers and wired connections. The Anhinga infrastructure provides a message broadcast ad hoc networking protocol and distributed computing platform based on Java Network technology [1] [2].

In this project, we introduce a new distributed computing paradigm called ***tuple board*** which is based on the *tuple space* abstraction originated by Gelernter [3] and the *blackboard* architecture popular in distributed artificial intelligence [8].

Most prominent implementations of tuple space systems including JavaSpaces [4] and T-Spaces [5] follow a centralized architecture where the space itself resides on a server, akin to a database server. Recently, researchers have attempted to develop decentralized architectures for tuple space systems. Some work includes LIME [6] and PeerSpaces [7].

In our opinion, the traditional implementation of tuple spaces is not well suited for ad hoc networks where devices frequently lose network access. The tuple space architecture attempts to provide persistent storage of tuples irrespective of the state of nodes in the network. While persistence may be easily implemented in centralized systems such as JavaSpaces, achieving this end in a purely decentralized environment would require complex replication schemes. This would place undue overhead on the resource-constrained devices that are typical for ad hoc networks.

In the architecture used in this project the availability of tuples is determined by the state of devices participating in the network. At any given point of time, only the tuples contained in devices that are active in the network are available on the tuple board.

As part of the project, a specific instance of the tuple board architecture was realized using the Anhinga infrastructure. The system, called ***Anhinga Board*** uses the M2MI protocol for the implementation of tuple board semantics.

A disciplined approach was used to build the Anhinga Board system.

The project constituted of the following phases:

- **Research**
  An in-depth literature survey was conducted investigating the design approaches of various tuple space systems. Salient systems studied included JavaSpaces [4], TSpaces [5] and LIME [6]. The survey indicated that most tuple space systems used centralized architectures. Davies, Friday et al [10] observed that the absence of multicast support in the past had limited the development of purely decentralized tuple space architecture. The M2MI system was chosen to address this issue. Also in this phase, the M2MI API was studied in detail and a few sample applications were developed.

- **Design**
  In this phase, various design options for the tuple board were evaluated and a prototype was developed. In parallel, the proof-of-concept convention center application was specified in detail. The final tuple board design is described in Sections 3 and 4.

- **Implementation and Testing**
  The tuple board system was implemented and the operations were tested using simple test harnesses. Following the successful testing, the convention center application was developed using the tuple board system.

- **Packaging and Report**
  Finally, detailed documentation of the system was put together, and the project report was written.

2 Overview of Tuple Board

A tuple board is a globally shared memory buffer that is organized as a collection of *tuples*. As in a tuple space, the basic element of a tuple board system is a *tuple*, which is a vector of typed values called *fields*. *Templates* are used to address tuples via *matching*. A template is similar to a tuple, but some (zero or more) fields in the vector may be replaced by typed placeholders called *formal fields*. A formal field in a template is said to match a tuple field if they have the same type. If the template field is not formal, both fields must also have the same value. A template matches a tuple if they have an equal number of fields and each template field matches the corresponding tuple field.

The table below shows some simple tuples and templates:

Tuple	Description	Matches template <float, “hello”, int>	Matches template <float, String, 20.0>
<1.2, “hello”, 20>	Fields: - float with value 1.2 - string with value “hello” - integer with value 20	Yes	No
<1.2, “hello”, 20.0>	Fields: - float with value 1.2 - string with value “hello” - float with value 20.0	No	Yes

A tuple board provides four operations: *post, withdraw, read* and *iterator*.

A *post* is a non-blocking operation that places a tuple on the shared board. A creator of a tuple can remove a previously posted tuple from the tuple board through the *withdraw* operation.

A *read* returns a posted tuple that matches a given template. If there is no match, the operation blocks until a matching tuple is posted. The iterator operation may be used to read a series of tuples from the tuple board. The operation takes a template as an argument and returns an `Iterator` object. Subsequently, read may be called on the iterator object to iteratively access a series of tuples. The operation only matches tuples that have not already been returned.

A tuple board provides a powerful mechanism for inter-process communication, which is pivotal to distributed programming. A process with data to share generates a tuple and places it on the tuple board. A process requiring data simply reads it from the tuple board.

Communication in a tuple board has the following characteristics:

- *Decoupled in space*: A Tuple board provides a globally shared data space to all processes regardless of machine or platform boundaries
- *Destination uncoupling:* In most message-passing systems the sender must always specify the receiver. The creator of a tuple, however, requires no knowledge about the future use of that tuple, or its destination, so tuple board communication is fully decoupled.

The key advantages of the tuple board system are below:

- *Flexibility*: A tuple board does not restrict the format of the tuples it stores or the types of data that they contain. Thus it can support the data needs of a broad range of applications.
- *Simplicity*: An application developer building a system with a tuple board would not have to manage point to point communication channels or deal directly with networking protocols. His focus would be on the data (tuples) that the application produces and consumes to get its work done. By reducing the communication management overhead, the tuple board approach simplifies the application development process vis-à-vis a message processing system.

3 Anhinga Board Architecture

Anhinga Board provides a lightweight tuple board implementation that does not rely on a central server to store the tuples.

The Anhinga Board API provides the following methods:

```
- post(Tuple t);
- withdraw(Tuple t);
- read(Tuple template);
- read(Tuple template, long timeoutInMS);
- iterator(Tuple template);
```

The following sections describe the architecture of the system and the above operations.

3.1 System Design

The design of Anhinga Board is intended to allow for collaboration without awareness of the participating group members, and aims to minimize the need for ‘always-on’ network access.

Each device in the network maintains a memory buffer where tuples are stored called the *tuples list.* Multicast communication using M2MI calls [2] are used to perform operations on the tuple board.

Each instance of the tuple board in the network has a unique ID, which is supplied by the application that instantiates the Anhinga Board layer. During a post operation, a tuple is tagged with the tuple board instance id and a sequence number maintained on the device generating the tuple. Subsequent to the post, the sequence number is incremented. When the device restarts, all tuple board instances running on it are restarted and the corresponding sequence numbers are reset to zero. Thus, every tuple on the tuple board is uniquely identified by the instance id and sequence number. There is a risk that the amount of storage, in this case 32 bits, may not be sufficient for the sequence number and could overflow eventually. However, we believe that in the scope of ad hoc networks of small devices, it is unlikely that a device would generate 2^{32} tuples without restarting, thereby mitigating this risk.

Applications interface with the Anhinga Board layer and are not aware of the underlying M2MI calls being executed to complete the operations.

Figure 1 shows a layered view of the architecture.

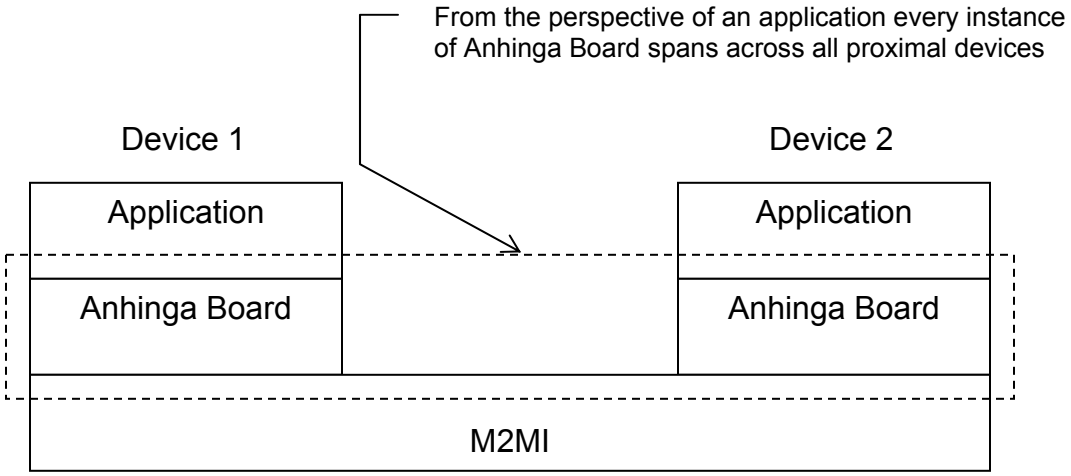


Figure 1: System Architecture

3.2 Design of operations

`post(Tuple t)`

A post operation simply places a tuple into the tuples list of the device.

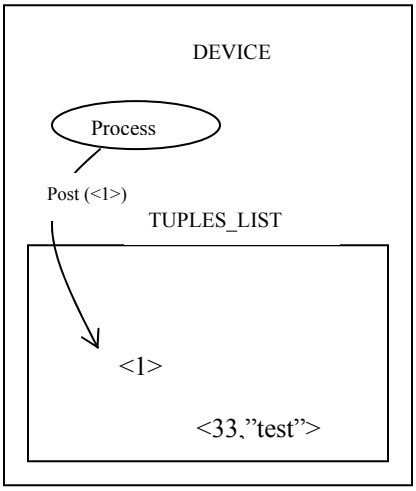


Figure 2: post operation

`withdraw(Tuple t)`

A withdraw operation removes a tuple from the tuples list in the device.

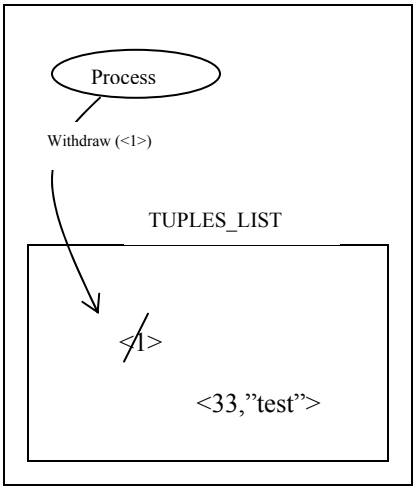


Figure 3: withdraw operation

`read(Tuple template)`

A read is a blocking operation that looks for a match for a given template by periodically polling all the nodes in the network.

In order to reduce the bandwidth used by this operation a list of hash values representing the objects constituting the template is multicast to the network. Since hashing functions are typically designed to reduce collisions, in most cases the tuples that are made of different objects will have different hash values. Multicasting the hash values in lieu of the template reduces the bandwidth consumption from a potentially large variable amount to a fixed number of bytes.

When a device participating in the network receives a read request, it searches its list for tuples that have the same hash values as the request. Any matches it finds at this stage are tentative as it is possible for two tuples with different objects to have the same list of hash values.

Subsequent to finding a tentative match, the device communicates directly with the requestor and passes it the tuple. The requestor verifies if the tuple is indeed a valid match by checking if the non-null fields of the template and the tentative match are equal. If it is valid match, the read operation is unblocked and the tuple is returned to the application. Otherwise, the tuple is simply ignored and the multicast is resumed. The timeout version of the method, `read (Tuple template, long timeoutInMS)`, will stop multicasting and unblock after the specified time interval even if a successful match was not found.

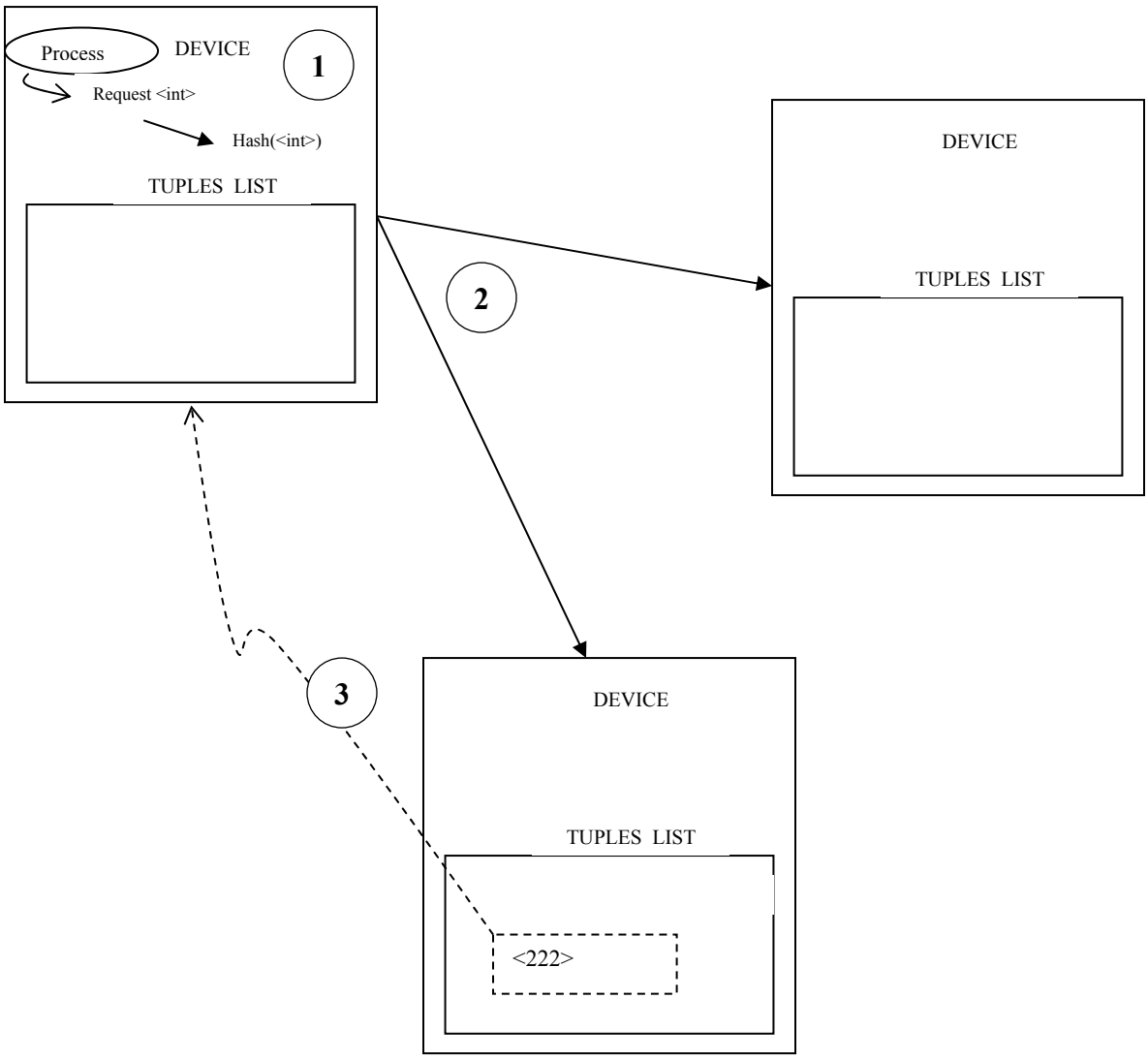


Figure 4: read operation

`iterator(Tuple template)`

This operator creates an `Iterator` object over a list of tuples that match the given template. The `Iterator` provides read methods that only return unique tuples to the requestor. A tuple is considered unique if it has a tuple board instance id or sequence number that has not already been received.

It is designed as follows:

1. As in the normal read operation, a hash representation of the template is periodically multicast to all nodes in the network.
2. The `Iterator` tracks the sequence number of the latest tuple received from each device in the network. The read request also includes this list of receipts.
3. A device receiving the request verifies if there are any new tentative matches, i.e. if its tuple board instance id is included in the request then tuples with larger sequence numbers are candidates for a match. If the tuple board instance id is not included, then all tuples in the device are valid candidates.
4. A tentatively matched tuple is sent to the requesting device. The requestor verifies if the tentative match is fully valid. It also updates the largest sequence number of tuples received from each device, irrespective of the match being a success.
5. When a full match is successful the operation stops the broadcast, unblocks and returns the tuple.
6. A subsequent read operation will restart the broadcast and follow the above steps to ensure that only new tuples (different tuple ids) are returned to the invoking process.

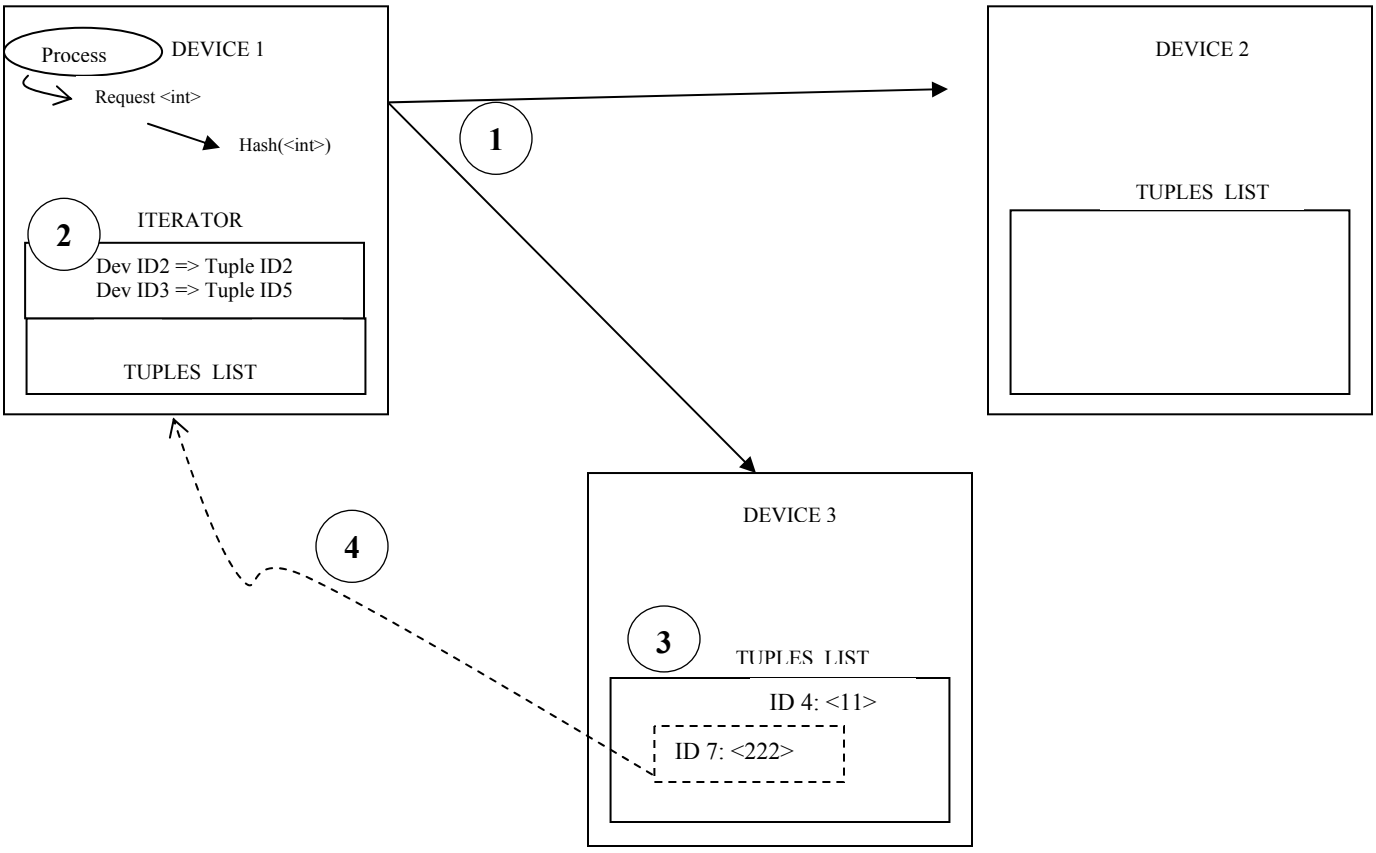


Figure 4: Iterator read operation

4 Anhinga Board Design

4.1 Class Descriptions

Figures 5 and 6 summarize the key classes in the Anhinga Board system. Each of these classes is subsequently explained.

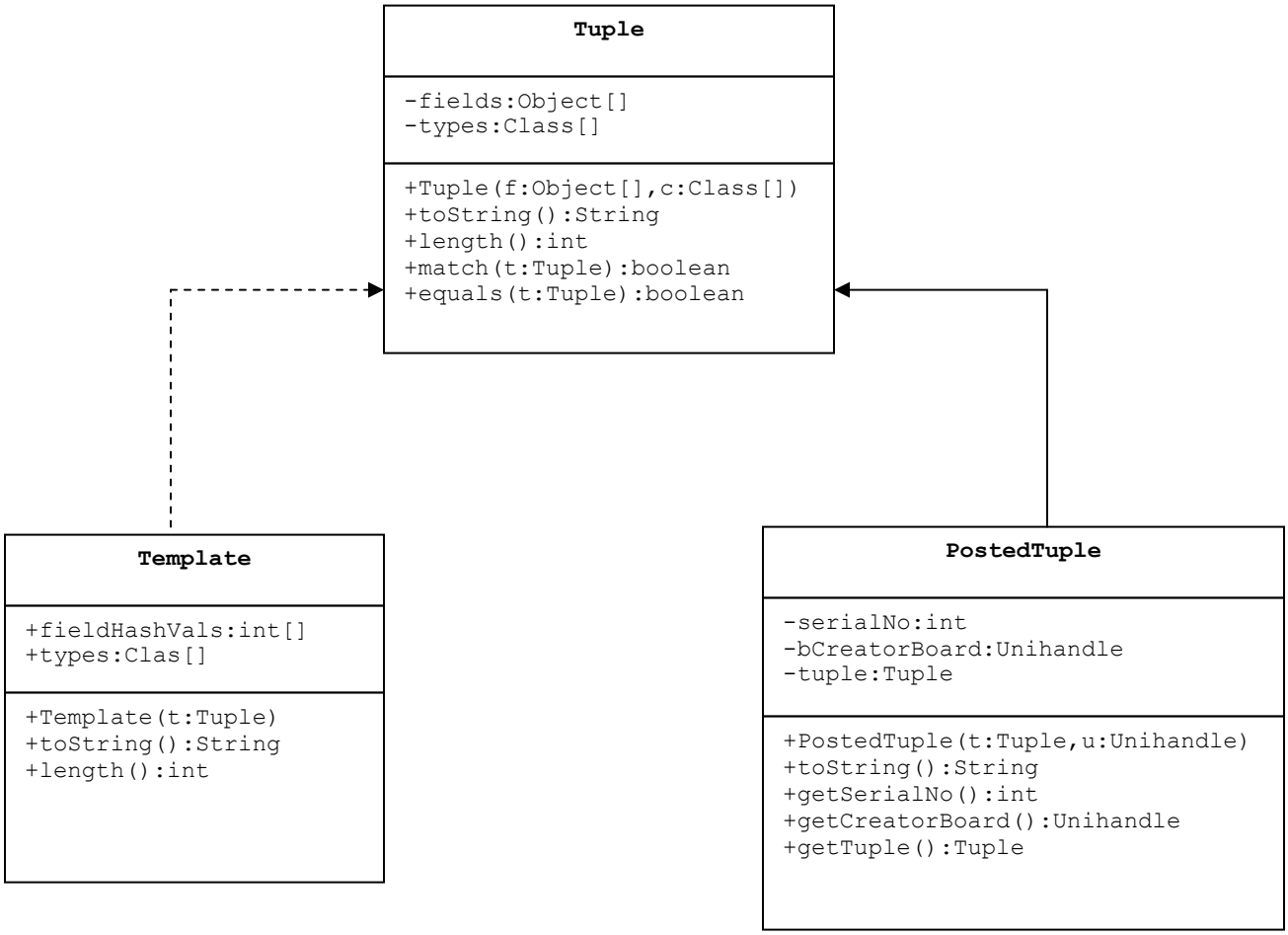


Figure 5: Anhinga Board Class Diagram

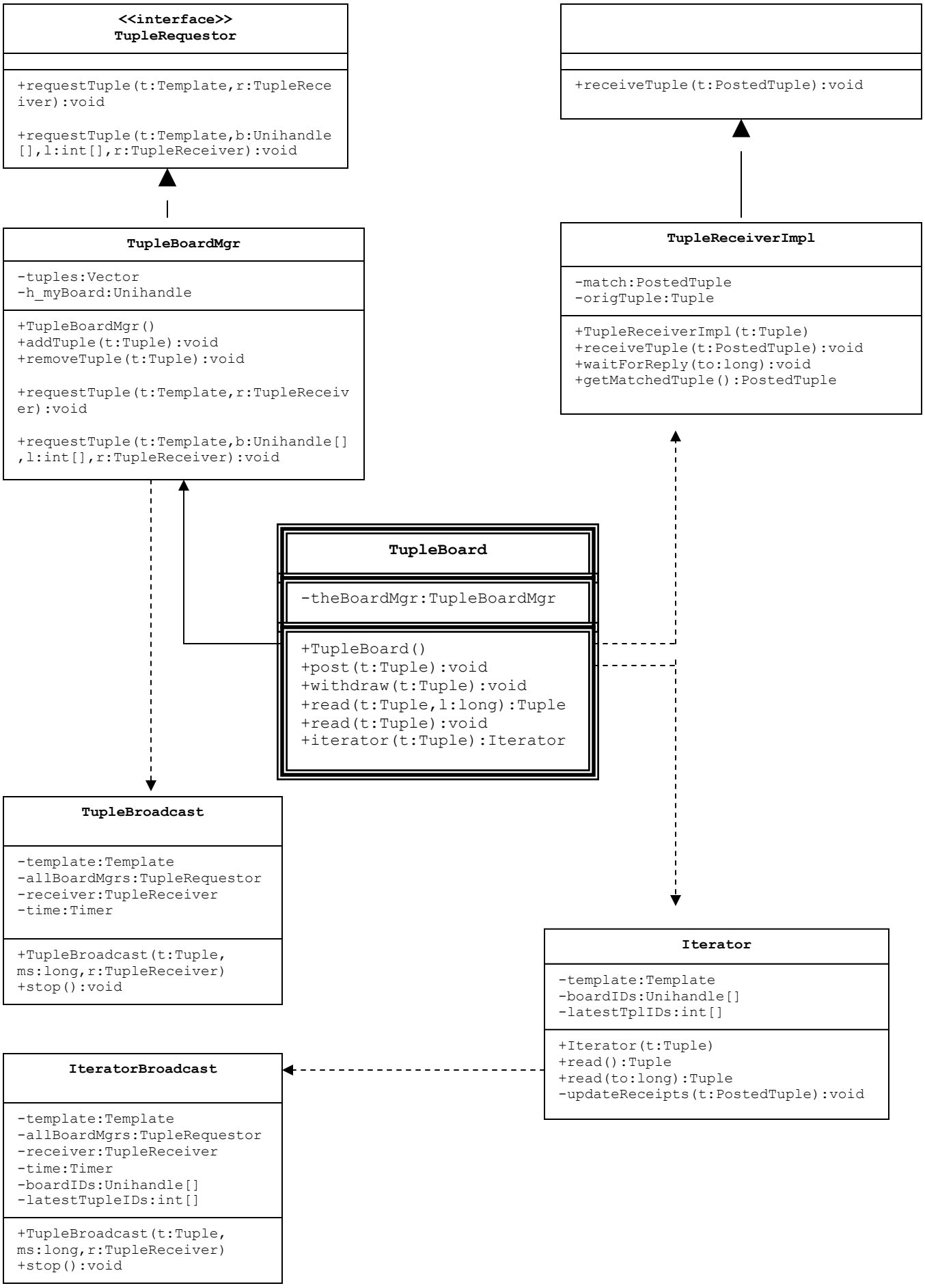


Figure 6: Anhinga Board Class Diagram

Tuple

This class represents a tuple as described in Section 2. Internally, it maintains a vector of fields and a vector of the types associated with the fields. In the implemented matching algorithm, the *match* method receives a template which contains a list of hash codes of the objects in a tuple. If all hash codes match then a tentative match is issued. It is the responsibility of the requestor to verify if the match is valid by invoking the *equals* method of the class.

PostedTuple

This class represents a tuple that has been posted on the tuple board. When a process posts a tuple on the Anhinga Board, the system tags it with a unique global identifier of the device where it was generated, and a sequence number (as described in Section 3.1). Every tuple in the system can be uniquely identified by these attributes.

Template

This class is used internally by the Anhinga Board system to reduce network bandwidth consumption. It maintains a vector of integers that correspond to the hash values of objects constituting a tuple.

TupleBoard

This class is the primary interface to the Anhinga Board system. It provides the tuple board operators described in Section 3 – post, read and withdraw. It also provides a factory method called iterator which can be used to create an `Iterator` object.

Interface TupleRequestor

This interface specifies the mechanism for requesting tuples from a Tuple Board over M2MI communication.

Interface TupleReceiver

Analogous to the TupleRequestor, this interface specifies the mechanism for receiving tuples from a Tuple Board over M2MI communication.

TupleBoardMgr

This class is responsible for interfacing with the M2MI layer for executing the tuple board operations. It manages the data that is local to each instance of a tuple board via the addTuple and removeTuple methods. It implements the TupleRequestor interface, thus receives and processes requests from other instances of tuple board via the requestTuple methods.

TupleReceiverImpl
This class implements the TupleReceiver interface and provides the mechanism to accept tuples from the Tuple Board.

Iterator
This class provides a mechanism to read a distinct set of tuples from the Tuple Board. An Iterator is not constructed directly, rather it is returned by a factory method, iterator(), of the class TupleBoard. The iterator will not match any tuple that had been returned on a previous call of its read method. If there is no tuple that matches the template, the read operation blocks until a match is found or a timeout occurs.

TupleBroadcast / IteratorBroadcast
These utility classes assist the TupleBoard class in broadcasting read requests in a background thread.

4.2 Anhinga Board Operation

This section describes how M2MI was used to design the Anhinga Board system.
The communication underlying all tuple board operations are driven by two interfaces – `TupleRequestor` and `TupleReceiver`.

Each device has an object called `theBoardMgr` that implements the below interface:

```
public interface TupleRequestor {
    public void requestTuple( Template template,
                              TupleReceiver reqFrom);

    public void requestTuple( Template template, Unihandle[] boardIDs,
                         int[] latestTupleIDs, TupleReceiver reqFrom);
}
```

`theBoardMgr` is responsible for handling all incoming requests for tuples that match a given template. The device also obtains from the M2MI layer an omnihandle for interface TupleRequestor, which allows it to communicate with all other devices in the network. Consequently, when attempting a read operation, a device invokes the requestTuple method on the omnihandle, and passes it the template, and a unihandle to an object that implements the `TupleReceiver` interface below:

```
public interface TupleReceiver {
    public void receiveTuple( PostedTuple tuple );
}
```

Figure 7 shows the sequence of M2MI invocations that occur when a device attempts to perform a simple read operation in the Anhinga Board system.

The device first calls `allBoardMgrs.requestTuple(template, receiver)` on an omnihandle for TupleRequestor. Since it is invoked on an omnihandle this call goes to all devices in the network. The invoking device now waits for potential matches for its request.
Each device that receives the request looks for a tentative match for the template, according to the matching scheme described in section 3.2. Upon finding a match it calls `receiver.receiveTuple(matchedTuple)` . The method is invoked on the unihandle passed in as an argument, and thus the call goes only to the requesting device and not to any of the other devices that may be present.

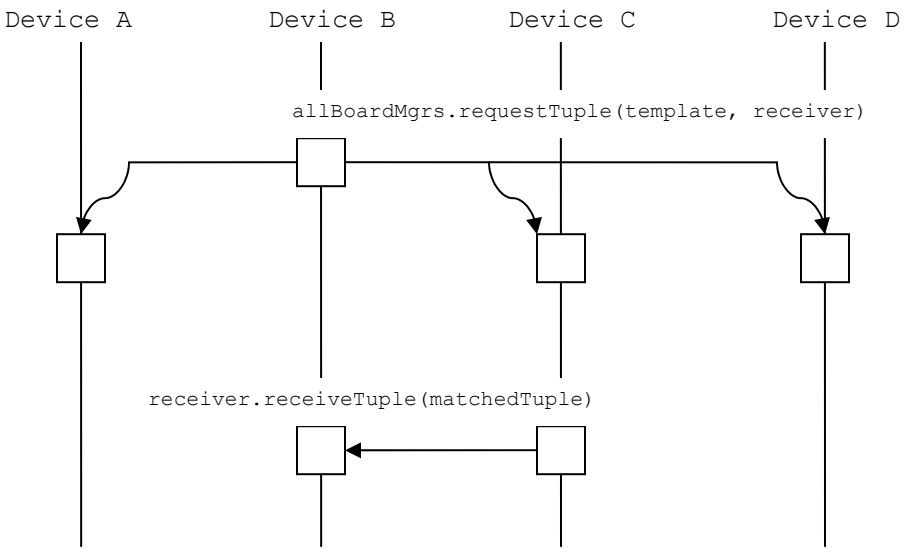


Figure 7: read Sequence Diagram

Figure 8 shows the sequence of M2MI invocations that occur when a device attempts to perform an iterative read operation in the Anhinga Board system.

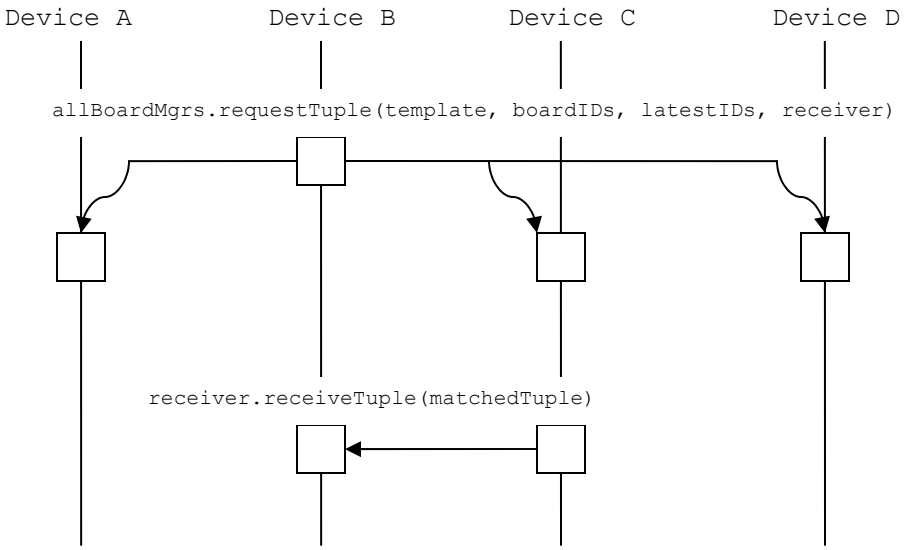


Figure 8: iterative read Sequence Diagram

As can be observed from the figure, the only change in this scheme is the arguments passed to the requestTuple method. The iterative read request includes two additional arguments – a list of tuple board instance Ids, and a corresponding list of largest tuple sequence numbers that were received from them. Thus, when a device receives an iterative read request, it only looks for matches with sequence numbers larger than the one it previously sent to the requesting device.

4.3 Implementation Samples

The code snippets below give an overview of an application's typical interaction with the Anhinga Board API.

The application should first initialize the M2MI layer and create a TupleBoard object as below:

```
M2MI.initialize(1234L);
TupleBoard theBoard = new TupleBoard();
```

The application can post a tuple on the board using the post method. For instance, the below code snippet posts the tuple {5, "testing"}.

```
Object[] fields = { new Integer(5), "testing" };

Class[] types = { Integer.class, String.class };

theBoard.post( new Tuple(fields, types) );
```

The application can read a tuple from the board using the read method. For instance, the below code snippet attempts to read a tuple that matches {5, null}, i.e. any tuple of length two, where the first value is the Integer 5, and the second value is any String object.

```
Object[] fields = {
                      new Integer(5),
                      null
                  };

Class[] types = {
                     Integer.class,
                     String.class,
                   };

theBoard.read( new Tuple(fields, types) );
```

In order to perform an iterative read, the application would create an Iterator object using the iterator method and invoke the read operation on the object.

```
Iterator itr = theBoard.iterator( new Tuple(fields, types) );
itr.read(); // reads first tuple
itr.read(); // reads second tuple
```

5 Anhinga Board Applications

Anhinga Board is expected to be a useful middleware for traditional collaborative applications such as groupware. We also expect it to enable collaboration in non-traditional contexts such as ad hoc networks in search and rescue.

For a groupware calendar scheduling application, each device could post tuples with the person's scheduled meetings on the tuple board, and then all devices can read the tuples and assemble a combined calendar showing every person's meetings.

Ad hoc networks are attractive in search and rescue situations, where traditional networking infrastructure such as fixed IP addresses may not already exist. Using Anhinga Board, devices may post tuples containing location information, broadcast distress signals, or conduct an interactive chat session.

As part of this project, an application targeted at convention centers and tradeshows was developed. The application is intended to enhance the experience of convention attendees by providing useful information at their fingertips. Moreover, Anhinga Board will enable peer-to-peer interaction at a level not supported by traditional client-server architectures.

5.1 Convention Center Functional Specification

The convention center application is geared to the needs of three key constituents:

- Visitors: People attending the convention
- Center: The site where the event is being hosted
- Exhibitors: Vendors exhibiting their products and services at the event

The application supports three bi-directional modes of interaction:

- Visitor-Center
- Visitor-Exhibitor
- Visitor-Visitor

Visitor-Center Interaction

Visitors will can look up information about the convention center and nearby businesses, which includes:

- Exhibit Hall Map
- Neighborhood Business Finder:
  Visitors may search for local business based on below criteria
  - TYPE – e.g. TAXI, RESTAURANT
  - PROXIMITY – distance from the center

The center itself can broadcast announcements to all users. This capability allows the center to notify visitors of any emergencies or other general announcements.

Visitor-Exhibitor Interaction

Visitors can pull up an *exhibitor list*, from which they may drill down to a detailed view of a specific exhibitor.

The detailed view consists of:

- Vendor Description
- Location / Booth Number

Additionally, an exhibitor can post advertisements on the anhinga board, which the application will periodically display to the visitors. A visitor may set up filters of varying granularity to screen these ads.

The criteria on the Ad Filter are the following:

- Vendor Name – e.g. IBM
- Industry – e.g. STORAGE

If no criteria are set then all ads will be delivered to the visitor. Otherwise, only the ads the match the filter criteria will be displayed.

Visitor-Visitor Interaction

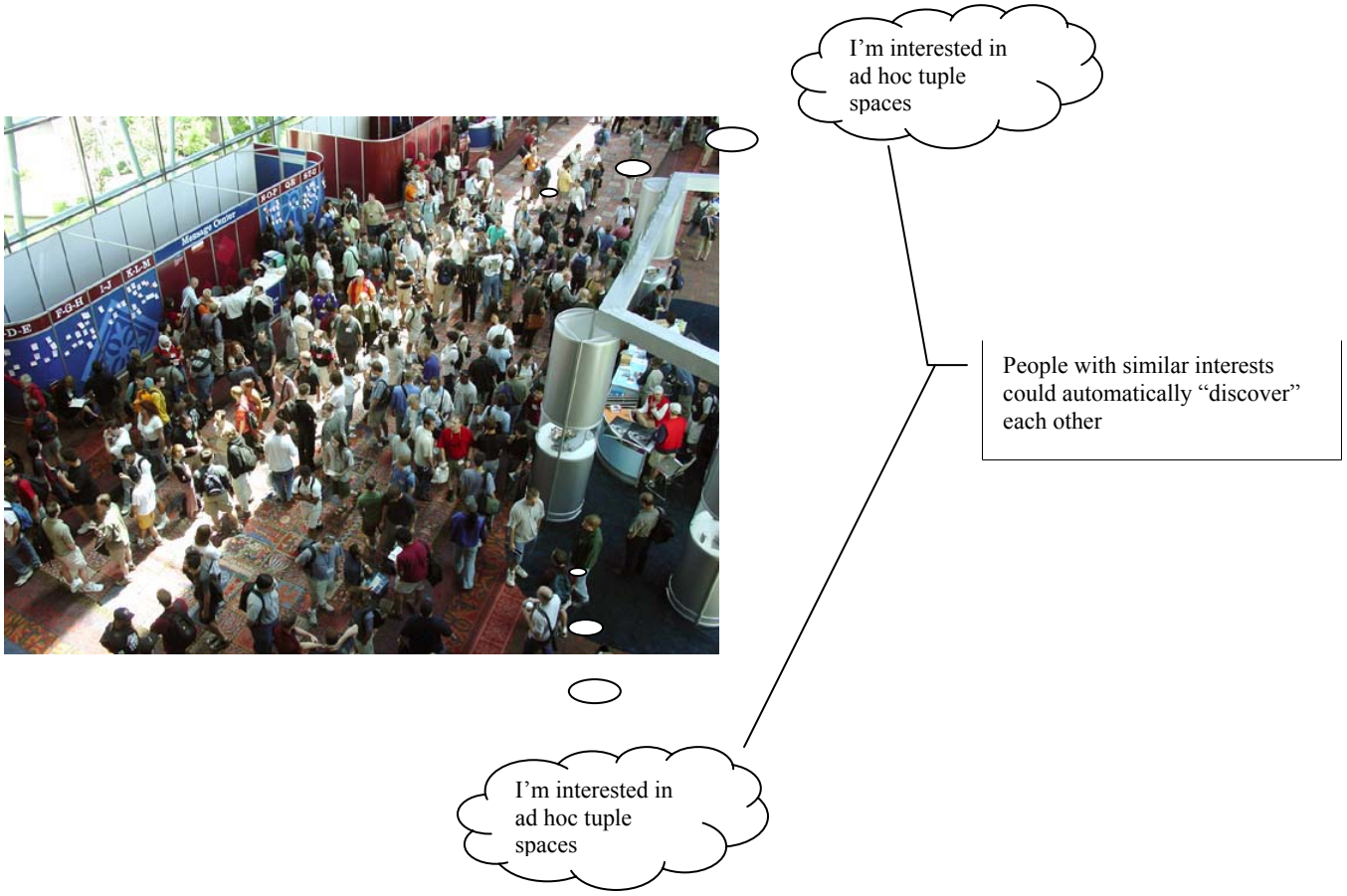
The application exploits Anhinga Board's capabilities in providing peer-to-peer interactions.

Convention attendees typically like to network with peers who share similar interests. However, there are no efficient means of discovering worthwhile networking opportunities. Using the application based on anhinga board, visitors can automatically discover other attendees of interest.

A visitor may enter criteria for discovering others, based on:

- Company
- Role
- Interests

When there is a match on any of the above criteria, it is displayed to the visitor who may initiate an interactive chat session with the new contact.



5.2 Convention Center Application - Design

The key consideration when designing an application based on the tuple board architecture is the structure of the tuples that are posted and subsequently read.

In the convention center application, each tuple includes a string designating its originator and another string designating its type, following which are an arbitrary number of objects.

The Venue and the Exhibitors *post* the tuples on the board, and the Visitors *read* the tuples from the board. In order to keep the user experience responsive the timeout versions of the read methods were used. This prevented any operation from blocking indefinitely.

Below is an overview of the major use cases implemented for each stakeholder and their corresponding user interfaces.

Venue Use Cases and Interface

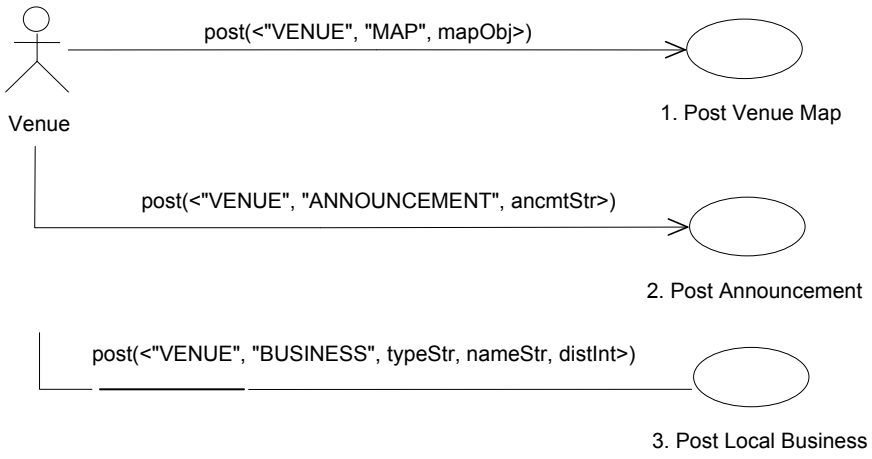
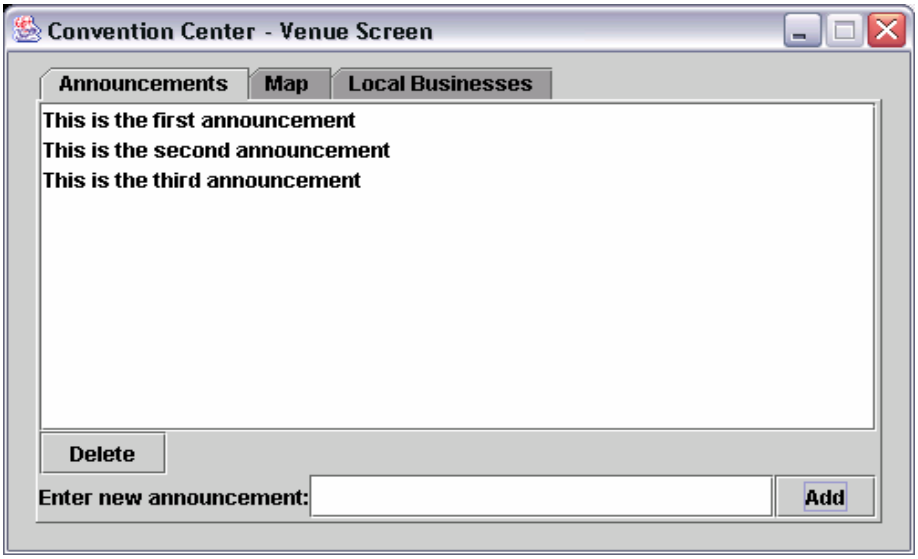


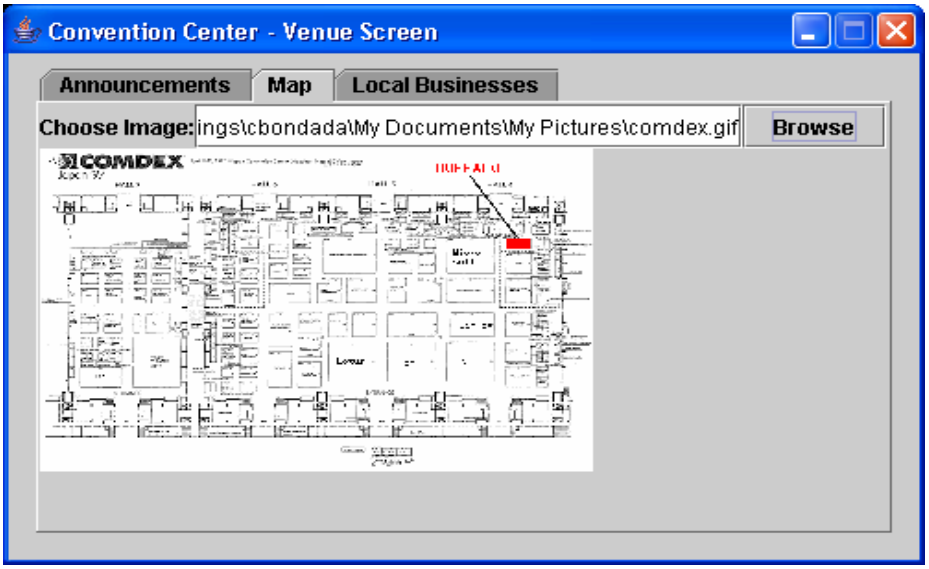
Figure 9: Venue Use Case Diagram

The Venue posts a map, by posting a tuple containing the string “VENUE”, the string “MAP”, and an object of type Image. Similarly, it posts an announcement by posting a tuple containing the string “VENUE”, the string “ANNOUNCEMENT”, and a string constituting the announcement itself. Finally, it posts information on a local business by posting a tuple containing the string “VENUE”, the string “BUSINESS”, strings constituting the name and type of the business and an Integer constituting the distance from the venue.

1



2



3

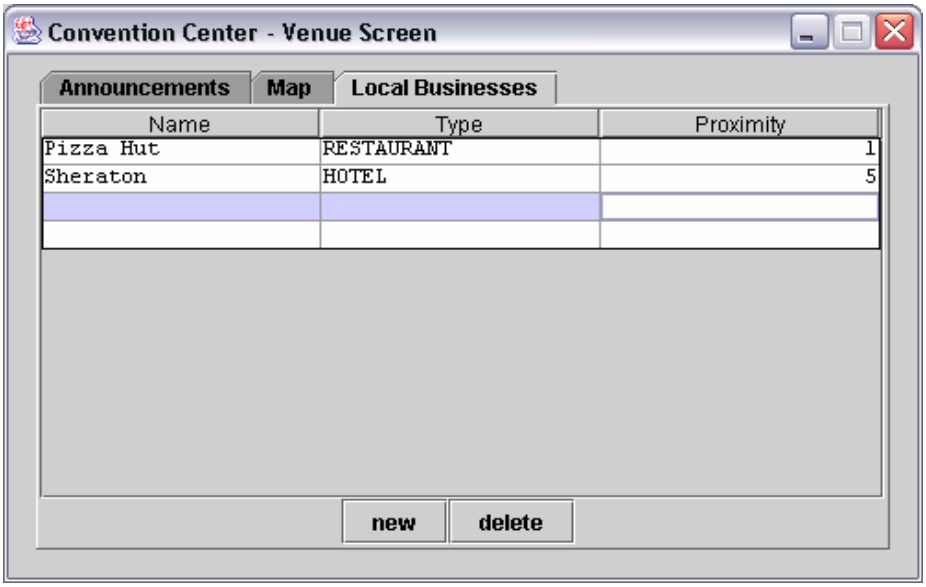


Figure 10: Venue Screen Shots

Exhibitor Use Cases and Interface

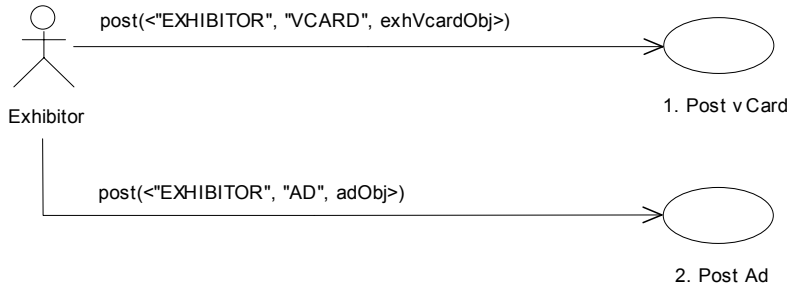
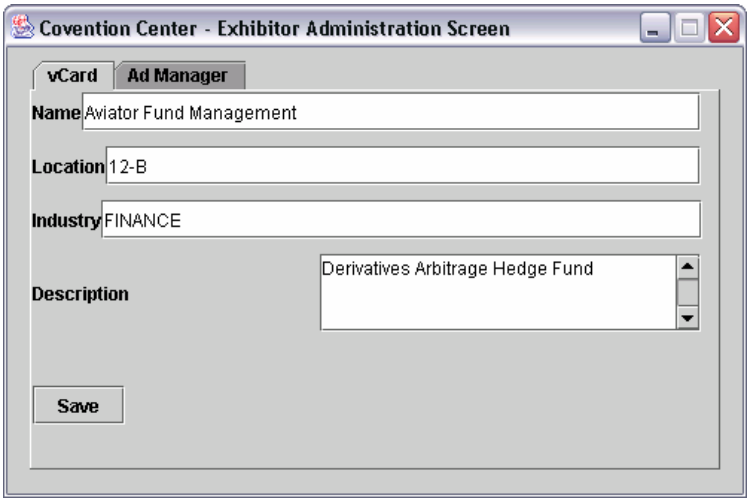


Figure 11: Exhibitor Use Case Diagram

An exhibitor posts its detailed information in the form of a visiting card. It posts a tuple containing the string “EXHIBITOR”, the string “VCARD”, and a custom object of type ExhibitorVCard. The class ExhibitorVCard contains strings constituting the name, location, industry and description of the Exhibitor.

Similarly, the exhibitor posts an advertisement by posting a tuple containing the string “EXHIBITOR”, the string “AD”, and a custom object of type Advertisement. The class Advertisement contains a string constituting the message, and a reference to ExhibitorVCard associated with the ad.

1



2

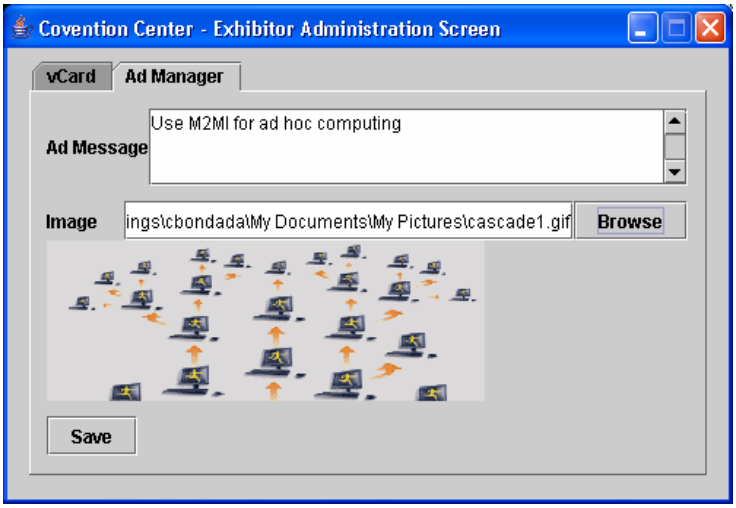


Figure 12: Exhibitor Screen Shots

Visitor Use Cases and Interface

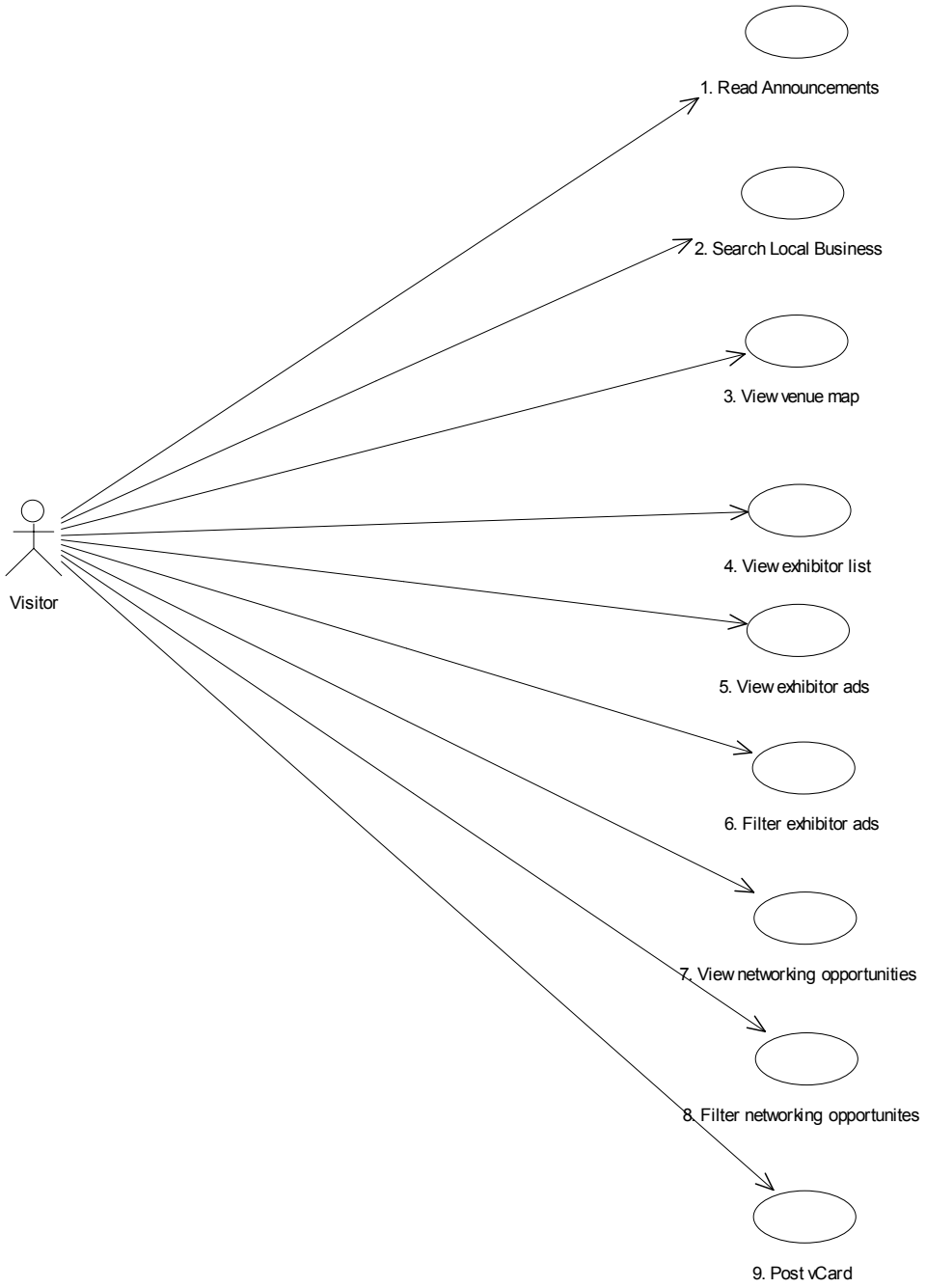


Figure 13: Visitor Use Case Diagram

A visitor is the primary consumer of data in the convention center application, and as such, typically performs read operations.

In order to read all announcements posted by the venue, the visitor program continually tries to find matches for the template {“VENUE", "ANNOUNCEMENT", null}. Similarly, the visitor looks up the map of the venue by matching {“VENUE”, “MAP”, null}.

The local business search functionality is user-driven, where the template is dynamically constructed according to the values entered by the user. For instance, a visitor searches for a hotel that’s a mile away by finding a match for {"VENUE", "BUSINESS", null, “HOTEL”, new Integer(1)}.

The program looks up all exhibitors with the template { "EXHIBITOR", "VCARD", null} and continually displays exhibitor advertisements that pass through its filter. It sends the filter criteria set by the user as part of the template it attempts to match. The template is made of the string EXHIBITOR, the string AD and an object containing the criteria. Subsequently, the program only receives advertisements that match the criteria.

A visitor optionally posts her detailed information in a tuple containing the string ‘VISITOR’, the string ‘VCARD’, and strings constituting name, title, company, interests and a custom object which is her instant messenger (IM) handle. Subsequently, other visitors will be able screen networking opportunities by attempting to match a dynamic template constructed from user-specified values.

A visitor may communicate with other visitors using an IM window. This simple instant messaging system was built using M2MI invocations as described in [2] to demonstrate that applications built on the Anhinga Board API are fully inter-operable with M2MI.

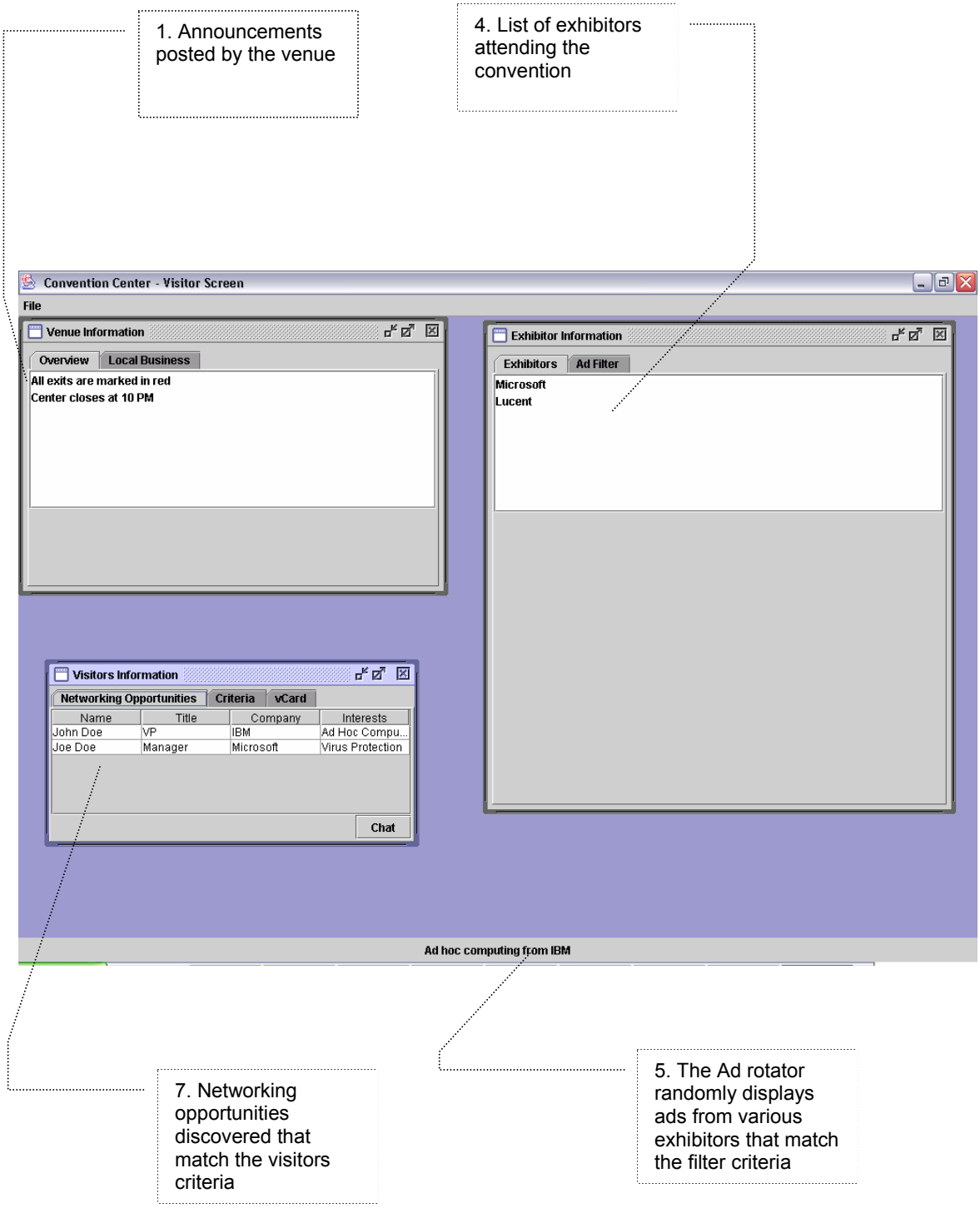


Figure 15: Visitor Screen Shots

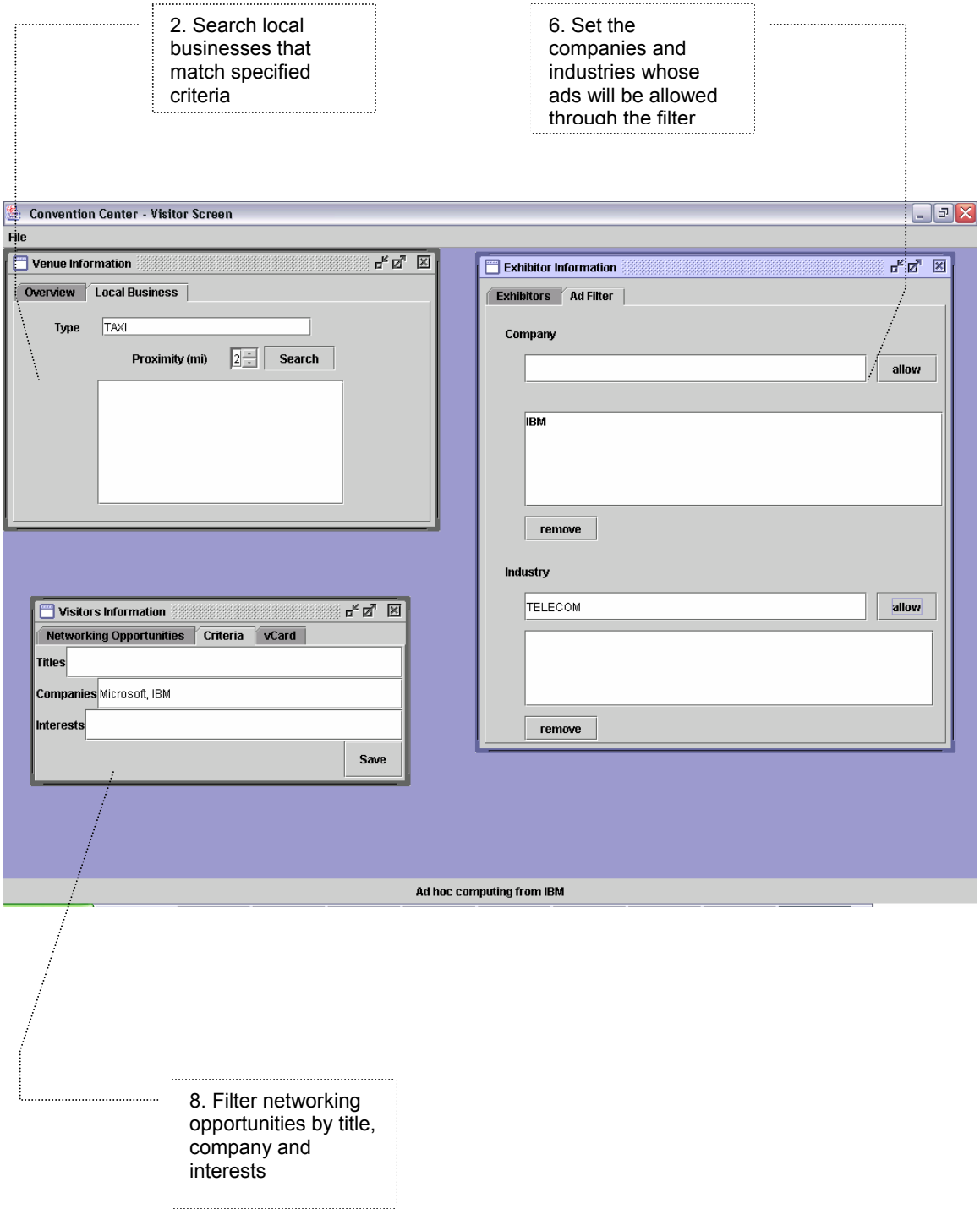


Figure 16: Visitor Screen Shots

5.3 Convention Center Application – Program Execution

This section describes the steps to setup and run the application.

Setup

1. Include in the **CLASSPATH** environment variable the following:

    - M2MI – The directory on the file system containing the M2MI library
    - tupleboard.jar – Contains the Anhinga Board API
    - convcenterapp.jar – Contains the Convention Center Application
    - AbsoluteLayout.jar – Contains UI controls developed by NetBeans [11]

Execution

1. Start the M2MPRouter as described in [1] and [2]. For example, use *java edu.rit.m2mp.ip.M2MPRouter* to run all processes on the same host.
2. Run each component of Convention Center Application as below:

    - **Visitor:** java convcenterapp/Visitor
    - **Venue:** java convcenterapp/Venue
    - **Exhibitor:** java convcenterapp/Exhibitor

6 Future Work

This project has demonstrated the value of the tuple board and M2MI systems. In the future there are several enhancements that can be added to the system to make it more full-featured and robust.

1. Tuple Persistence:

Currently, the system does not provide any persistence of tuples. All the replication schemes considered were simply not suitable for a decentralized system aimed at resource-constrained devices. However, tuple persistence could greatly improve the reliability of the tuple board system. Any persistence scheme should factor in processor, memory and bandwidth limitations.

2. Matching Semantics:

According to the current matching semantics it is not possible to match tuples where some of the constituting objects are not equal. For instance, if the developer wants one object of a tuple to be a successful match for another if *any* of their attributes are equal, as opposed to *all* their attributes being equal. Future work could explore different kinds of tuple matching semantics such as the match-any scheme mentioned above. Extending the matching mechanism with a querying language similar to SQL can be valuable in improving the developer-friendliness of the system

3. Withdraw Notifications:

Currently the system does not provide any notification of a withdrawal of a tuple. Thus, an application may be using a tuple that it *read*, but was subsequently *withdrawn*. A mechanism that allows applications to detect tuple withdrawals would be useful.

4. Transactions:

It would be worthwhile to provide transaction like capabilities. This could help improve the reliability of tuple board applications.

5. Security:

Currently all data communication is not authenticated or encrypted. Clearly, this is not desirable in a real world context. For example, in a military system non-authenticated users must be prevented from posting or reading tuples. The work proposed by Binder [9] may help address this issue.

7 Conclusions

The Anhinga Board API was successfully developed using the M2MI protocol. The M2MI paradigm proved to be an ideal foundation for building the decentralized architecture of the tuple board system.

The Anhinga Board API simplifies the development of distributed applications. Both client-server and peer-peer applications can be easily realized using the tuple board. From an application developer's perspective, the API removes significant overhead of managing the network infrastructure and processing code, allowing the developer to focus on the application logic.

The server-less architecture of the tuple board paradigm has significant advantages

1. There is no single point of failure in the system. Unlike a database system, the failure of any single node will not bring down the entire system. The system would be limited, in that the tuples previously posted by the failing node will no longer be available. However, all other nodes will still be able to interact.

2. The tuple board architecture reduces the overhead of server maintenance and monitoring. Significant administrative effort is extended in a typical client-server system towards ensuring that the server is operational, has sufficient physical resources such as memory and disk space, and in general is not a bottle-neck. By shifting storage and processing to the participating nodes, the tuple board system would require less maintenance and monitoring as compared to a client-server system.

The convention center application developed in this project clearly demonstrates the value of the peer-peer architecture inherent in tuple board paradigm. The following points highlight some observations of using the tuple board API.

1. When building an application using the tuple board, it is helpful to think of the data that each system component generates and consumes. These directly translate into the tuples *posted* and *read* by the component respectively.

2. Following which, the development of the components is simple as the data access needs are directly met by the tuple board primitives.

3. Since the tuple board primitives are blocking operations, the application could invoke them in background threads so that the primary thread remains unblocked. This is applicable in GUI applications, where the GUI should continue to remain responsive while performing time-consuming read operations.

8 References

[1] The Anhinga Project. http://www.cs.rit.edu/~anhinga

[2] Alan Kaminsky and Hans-Peter Bischof. Many-to-Many Invocation: A new object oriented paradigm for ad hoc collaborative systems. *17th Annual ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA 2002),* Onward! track, Seattle, Washington, USA, November 2002.

[3] D. Gelernter. “Generative Communication in Linda”, *ACM Transactions on Programming Languages and Systems*, Volume 7, Number 1, pages 80-112, January 1985.

[4] E. Freeman, S. Hupfer and K. Arnold. “*JavaSpaces: Principles, Patterns, and Practice*”, Reading, MA: Addison-Wesley, 1999.

[5] P. Wyckoff. “T Spaces”, http://www.research.ibm.com/journal/sj/373/wyckoff.html

[6] G.P. Picco, A.L. Murphy, and G.-C Roman. “LIME: Linda Meets Mobility”, In Proc. Of the 21st International Conf. on Software Engineering, pages 368-377, May 1999.

[7] Busi et al. “PeerSpaces: Data-driven coordination in Peer-to-Peer Networks”, In Proc. Of the 2003 ACM Symposium on Applied Computing.

[8] D. Corkill. “Blackboard Systems”, *AI Expert* 6(9):40-47, September 1991.

[9] Hans-Peter Bischof, Alan Kaminsky, and Joseph Binder. A new framework for building secure collaborative systems in ad hoc network. *Second International Conference on AD-HOC Networks and Wireless (ADHOC-NOW '03),* Montreal, Canada, October 2003. Accepted for publication.

[10] Davies et al. “An Asynchronous Distributed Systems Platform for Heterogeneous Environments”, In Proc. Of the 8th ACM SIGOPS European Workshop: Support for Composing Distributed Applications, Sintra, Portugal, ACM Press.”

[11] NetBeans. http://www.netbeans.org