AppBus: Mobile Device Application Collaboration via Short Term Memory

Craig Janssen
Motorola, Motorola Labs, Schaumburg, IL, USA
craig.janssen@motorola.com

Michael Pearce, Shriram Kollipara and Nitya Narasimhan
Motorola, Motorola Labs, Schaumburg, IL, USA
Email: {michael.pearce, shriram.kollipara, nitya}@motorola.com

***Abstract*—The increasing quality of computation and connectivity on mobile devices has motivated a need for data-sharing between resident services. Most such services rely on the user to transfer data manually between them. This approach is not only prone to input error but also creates a significant data-entry burden for the user on a mobile device. With AppBus, we envision a cooperative data-sharing facility that provides the equivalent of a "short-term memory" for the mobile device. By leveraging this transient data store and event notification system, services can collaborate in a loosely-coupled manner that makes their combined operation appear more intuitive and seamless. In this paper, we describe the design and implementation of AppBus for a mobile phone and initiate a discussion on both the utility and the limitations of this paradigm in mobile computing.**

***Index Terms*—collaborative application frameworks, event notification, mobile devices, IPC, associative data**

I. Introduction

As mobile devices become more feature-rich, we will see increased computation and connectivity capabilities on the device that will drive a new breed of collaborative applications. At the same time, portability requirements will ensure that some constraints (e.g., small display size, limited input capabilities) are likely to remain as challenges for mobile applications.

This causes an interesting dilemma for mobile application developers. The convergence motivates a need for increased data-sharing between services on the device. However, most of these services are developed independently and need additional 'bridging' mechanisms for such collaboration. Consequently, developers may be forced to take one of the following approaches:

a) *Implicit Sharing By Integration*. Services share data by being tightly-coupled into one "suite". Data can be shared within that suite, but is not visible to other services.
b) *Explicit Sharing By User Action.* Reliance on the user to transfer data manually. Applications exploit the user's short-term memory to create the context bridge between services.

Both approaches have limitations. Integration not only requires prior knowledge of the applications that need to collaborate, but it effectively limits the reuse and extensibility of the data-sharing mechanisms outside the suite. On the other hand, the user action approach places a non-trivial data-entry burden on the user, which is further exacerbated by the inherent input and display constraints of mobile devices. As a result, such mechanisms may suffer from user input error and can degrade the user experience. Consequently, a driving decision behind developing the AppBus framework was to design a solution that was more intuitive to both the user and the application – enabling data-sharing between decoupled applications with minimal user involvement or knowledge required.

In this paper, we discuss the goals, design and implementation of the AppBus framework with some emphasis on its use in mobile phones. Section II introduces the AppBus project, including some use cases that motivated the effort. Section III provides a more detailed look at AppBus components and the roles that they play to create the whole AppBus as well as some information on current implementation status. Section IV discusses security considerations in shared data exchange and our current method of balancing improved security against the freedom of applications to collaborate. Section V examines the usage and applicability of AppBus including a revisit to a previous use case, now AppBus enabled, as well a discussion of application communication complexity. Section VI discusses the relative contextual relevance of short term memory and long term memory. Section VII looks at related technologies that contributed to our vision for AppBus.

II. The AppBus Framework: Overview

We begin by describing one motivating usage scenario, followed by the design goals and operation.

A. Example Usage – Travel Planner

A user needs to make arrangements for an upcoming business trip using a mobile device. This will involve

booking flights, reserving a hotel room and making a dinner reservation.

Using currently available methods, the completion of this task may be accomplished in one of two ways. The first method involves several different applications on the phone and the user is the data mule between them.

A flight reservation application is activated and the user enters in all of the data including airline, flight times and destination. Upon reservation creation, the user copies data to the clipboard, closes this application and activates the hotel reservation application.

A hotel reservation is made with the user doing a paste of some previous entries (travel dates, location). The remaining information is entered and the user completes the reservation. Once again, the user copies data (date & time, hotel location) to the clipboard and closes the application.

The restaurant application is activated and the user does another paste of information. A nearby restaurant is found and reservation made.

While the use of a clipboard requires explicit user action, the absence of it requires even more user action in the form of reentered data. The workflow is further complicated if there are problems requiring backward steps, such as unavailability of a hotel room on the given day causing a change in airline reservations.

As is illustrated by this example, the first method's approach to the task is to use repetitive user actions and the user's short term memory to transfer data between the applications.

The second method that might be available is an integrated application combining all of these functions. This might eliminate some of the explicit data entry but at the cost of increased resource consumption and possible loss of flexibility and user choice.

For instance, in order to get an application that does airline, hotel and restaurant reservations, the user might be required to carry along the additional functionality of a movie theater locator and a car rental application. Not only is the additional functionality unneeded, but may waste screen and computational resources.

Or perhaps the user prefers Continental's flight and Marriott's hotel reservation applications. They may provide for less manual data input and/or preferential pricing. Using the best of the breed point solutions may not be possible with an all inclusive, generic application.

Either way, current solutions come up short and the user is the one picking up the slack or losing out.

B. AppBus Design Goals

In our approach to AppBus, we had several design goals:

Compact - Implementation that is compact enough to fit on a limited resource device

Intuitive - Minimal amount of direct user interaction required to enable its operation

Versatile - Shared data format that is versatile enough to convey a wide variety of information

Relevant - Data is retained by the AppBus for a useful period of time

Synchronous - Ability to query the communal dataset to retrieve information matching provided specifications

Asynchronous - Subscribing applications can be notified of significant data events

Spatially decoupled - No direct application to application communication or direct knowledge of other applications required

Temporally decoupled - Applications do not need to be active at the same time to share data

Autonomous - Automatic garbage collection of unused shared data

The desired effect of these goals was to have a system that encouraged application collaboration without direct user involvement, enabling us to create a short-term memory on the device rather than relying solely on the one in the user's head. This would allow multiple smaller applications to act more like a comprehensive whole without having the overhead of a large, monolithic application.

It may be noted that other currently available technologies fulfill various aspects of this goal list. However, none were found that completely fulfilled the goal criteria. Nevertheless, many of them provided their own useful design insights. Some of these technologies are summarized in section 5 along with their relative pros and cons with respect to our context.

C. AppBus Basic Operation

Some basic principles of operation of the AppBus are as follows:

Inserting Data

1) Shared data is placed on the AppBus in the form of a Data Object. A Data Object encapsulates a base piece of data and an unlimited number of attributes. The attributes are name – value pairs

2) Any application may update the information in any data object on the AppBus, regardless of which one originally inserted the Data Object.

Synchronous Data Retrieval

1) The AppBus framework may be queried to retrieve a set of Data Objects that match specific criteria (i.e., attribute values).

Asynchronous Data Retrieval

1) Applications may register with the framework to be notified of the insertion of new AppBus Data Objects.

2) Applications may also register with individual Data Objects to be notified of changes to their contents.

Organizing Data

1) Data Objects on the AppBus may have Associations created between them, implying a useful relationship. Data Objects may belong to multiple Associations.

Administration

1) AppBus Data Objects are removed from framework indexing based upon their date of creation or last update (oldest Data Objects are purged). However, any application that retains a direct reference to the object may continue to use it indefinitely.

III. AppBus Design and Implementation

From a design standpoint, there are three major roles played in the AppBus environment. These are: the Application, the Data Object and the Framework. A conceptual view of these elements and their interactions can be seen in Figure 1.

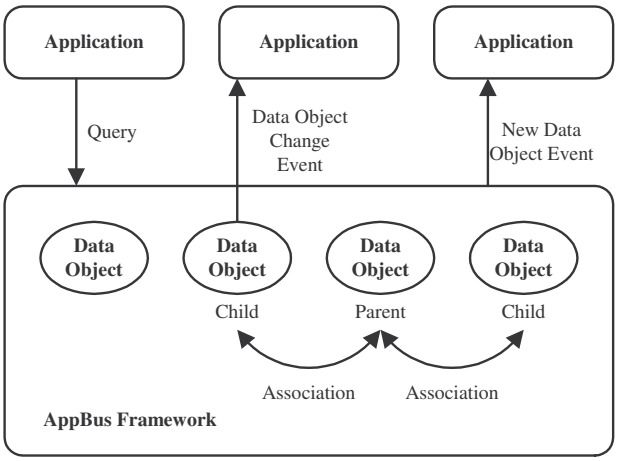

Figure 1. AppBus Framework

A. Data Object

Data Objects are the units of information that are placed onto the AppBus to be made available for sharing. Data Objects encapsulate a base piece of data called a payload, which may be of any type, and descriptive attributes, which may be of type text string, numeric or date. There is no fundamental reason that the attributes are restricted to these types, other than a bound set of types must be established in order to enable querying and sorting. If required, new types can be added in the future. The attributes are intended to provide contextual information about the payload contained within the Data Object. It is also possible that the collection of attributes in the Data Object represents the whole of the contained information, with no payload. There is no requirement for a payload to be present.

Data Objects may be organized into Associations within the AppBus. An Association is an informal linkage that Applications create between Data Objects to indicate a relationship between the data contained within the associated Data Objects. A Data Object may have one of two roles within a relationship: parent or child. A Data Object may have multiple parents or multiple children (and, correspondingly, this allows them to have siblings). However, a Data Object may not fill the role of both parent and child at the same time. This makes for a flat, two level hierarchical relationship tree that clusters around parents. This restriction serves two main purposes:

1) *Ease of purge implementation* - When the data objects are not allowed to form into arbitrarily large, very complex webs, it is easier to identify those Data Objects whose relevance is diminished and are good candidates for garbage collection in order to free space and keep the AppBus content at a more manageable level.

2) *Semantic relevance* - If every data object is allowed to simply associate with every other data object, the whole of the data encompassed by the AppBus may soon become one big generic grouping that provides no additional contextual benefit over unassociated Data Objects. Conversely, if the associations are formed with strict hierarchical relationships, loose inter-application collaboration is more difficult since all the applications must structure their data in the same manner.

Data Objects may be modified at any time while they are on the AppBus. This includes changes to Attributes, base data (payload) and Associations. Any time a change is made to a Data Object, its timestamp is updated in order to let Applications know how "fresh" it is. In addition, any application that subscribes to the state of a particular object will be notified of any modification.

B. Application

Applications may be both producers and consumers of the information available on the AppBus. An application fills the role of data producer by asking the AppBus Framework to create a new, blank Data Object for it, then filling in the relevant base datum and attributes, and finally submitting it to the AppBus Framework for insertion. Once it is inserted on the AppBus, the Data Object becomes visible to the other Applications using the AppBus and is available for their use.

However, creating new Data Objects is not the only way for an application to be a data producer. The AppBus is designed to be used for unstructured collaboration. Therefore, an application may modify or update Data Objects previously created by other applications or create associations between existing Data Objects.

An Application may fill the role of data consumer in one of two ways. The first way is by using the AppBus in a synchronous manner and querying the AppBus Framework for result sets of Data Objects that match certain defined criteria. In this case, the Application will provide names for the attribute to match and values that correspond to the named attributes. In this way the application may get desired information on demand.

The second way the Application may fill the role of consumer is to register for asynchronous notification of data events on the AppBus. The Application may register with the AppBus Framework for new Data Object insertion events, or it may register with individual Data Objects for notification about changes in their internal state. Registration entails providing the Framework or the Data Object a callback method to use for event notification.

C. Framework

The Framework is the overall structure that indexes the Data Objects. It is where an Application goes when it wants to perform Data Object queries, create Data Object associations, and find the members of a particular association or register to receive notifications of new Data Objects being inserted onto the AppBus.

Queries can be made on the basis of the existence of an attribute, the value of a single attribute or the values of multiple attributes. Matches can be requested for

attribute values equal to, greater than and less than. This is the primary reason for restricting attribute types: so that greater than / less than have meaningful results in performing queries.

As stated previously, Applications may register with the Framework to be notified of new Data Objects being added to the AppBus. In this way, a given Application may screen all Data Objects being added to the AppBus in order to watch for particular ones of interest.

The Framework is also responsible for purging the AppBus. Purging is done on a Data Object "freshness" basis (comparison of timestamps) and is integral to how the AppBus creates the idea of a "short-term memory". Unlike some other data repositories and message passing methods (such as a Tuplespace or Named Pipe), a Data Object does not have to be read or retrieved in order to be removed from the AppBus. If it does not get updated or "refreshed", it simply ages, becomes less relevant, and is eventually removed.

Purging is one of the places where Data Object associations become important. The purge routine removes the oldest Data Objects from the AppBus. However, it is assumed that if Data Objects are associated together, they provide contextual reference for each other and should be managed as a group. Therefore, families are purged together. A parent is removed along with all of its children that have only this one parent. If, however, a child has more than one parent, then only the association to the purged parent is removed. The child itself will remain on the AppBus to be purged in the future, along with one of the other association groups of which it is a member.

D. Implementation

We have implemented and evolved AppBus on a several different platforms. The first generation was written in Java and ran on a PDA. As an exercise in the proof of compactness and portability, this version was later ported to Java MIDP running on a Motorola mobile device.

The second generation was written in C# and running under the .NET Compact Framework environment associated with the Microsoft Windows Mobile platform. The hardware devices we deployed it on include the Motorola MPX[1] and the Hewlett Packard iPAQ mobile devices.

The third and current generation is written in C++ and runs on Linux based Motorola phones. The phones currently being used for implementation are the Motorola E680i[2] and the Motorola A1200[3].

Implementation migration was driven by the optimal resources available on the different targeted hardware platforms. As this migration across platforms occurred, various issues needed to be addressed. Sharing data across process address spaces is one. This was less of an issue in Java where all of the communicating programs were operating in the same Virtual Machine. Threading is another important aspect that required reexamination. In order to allow for the appropriate level of responsiveness to the applications, separate threads are required for event notifications, framework maintenance, etc. The threading API in C++/Linux is quite different from that used in Java and both were different from that used in C#.

IV. SECURITY

Any environment where data is being shared is ripe for exploitation by malicious applications. Therefore, security measures must be put in place to limit the ability of such applications to perform serious damage. However, AppBus is designed explicitly for unstructured collaboration. Any restrictions placed on the creation of data or access to created data reduces the ability of applications to collaborate in unstructured, non-predefined ways. If applications can only create and view their own data, no collaboration is possible. However, in an environment where services cannot be accessed and no data can be trusted, applications cannot collaborate either. Therefore, it must be determined where to set the mark between those two extremes.

A. Classification of Security Breaches

In our process of adding security, we classified security breaches into three categories:

1) Leakage of confidential information to unauthorized parties

This occurs any time an application accesses data and uses it for non-legitimate purposes. Data Objects on the AppBus may be queried for and read by any attached application.

2) Corruption of correct data by rogue applications

This can happen in two ways. The first method is an application may access an existing Data Object with legitimate data and alter its contents. Any Data Object may be updated and changed in any way by any attached application. Attributes may be changed or removed, new attributes may be added or the payload may be changed, removed or added by any application. The application does not have to be the creator of a Data Object in order to have complete access to it. The second method is an application may create a new Data Object with similar data and attributes to an existing Data Object but with differences that corrupt its purpose. Since this new Data Object is more recent than the previous one, it may be viewed as more relevant or as an update to the previous (legitimate) one. Any application may create new Data Objects on the AppBus.

3) Denial of service to legitimate applications through abuse of services by rogue applications

Rogue applications may create many new bogus Data Objects in an attempt to drown out the legitimate ones. This differs slightly from the corruption of data method above in that the rogue application is not attempting to simulate legitimate data, but rather simply drown it out with brute force. Or, the rogue applications may make unending modifications to existing Data Objects in an attempt to overload the asynchronous update mechanism. They may attempt to register for asynchronous notification of updates on Data Objects and consume all

the CPU time allocated to the update notification thread. They may conduct endless, meaningless synchronous queries on the AppBus framework in order to consume CPU time and over exercise thread synchronization locks in the AppBus.

B. Security Implementation

The methodology initially chosen to provide a measure of security in the AppBus is the fundamental one of sandboxing. Sandboxing is a method that restricts the activities of specific applications based upon the trust level assigned to the application. This is a method that has been commonly used in various places, including the Java VM.[4]

The sandbox implementation on AppBus is as follows. There are two separate data areas in the AppBus framework – an open one and a secure one. The application presents credentials upon initial connection to the AppBus. Based upon these credentials, the AppBus framework associates a security level with the connected application which allows it access to the secure area, the open area, or no access at all.

Applications with access to the secure area may also access the open area and may choose to perform operations (such as queries) on either area or both combined. Similarly, applications with secure access may choose to receive asynchronous update notifications from the secure area, the open area or both. The separate areas have separate resources associated with them – data stores, notification queues, computation threads, etc. so that problem activities in one area have minimal impact upon the other area. Of course, the expectation is that the problem activities will be in the open access area.

Data objects are labeled at their initial attachment to the AppBus whether or not they are secure. This classification is based on which area (open or secure) the creating application requests the Data Object be inserted. Of course, an application may only insert a Data Object into any area for which it is authorized. This classification may not change at any time in the Data Object's lifetime.

The reason for the classification lockdown is due to concerns about information leakage and data corruption. Assume that a Data Object could change its secure status. Then it must be an application with secure credentials that makes the determination to change its status. (The framework knows nothing about the meaning of the data it indexes.)

Changing a Data Object from secure to open runs the risk of exposing data that needs to remain secure. Regardless of whether the application doing the moving was the one that created the Data Object and has decided that the data it posted no longer needs protection, other applications may have modified it. Even if the application can prove that no other applications modified the Data Object, other applications may still have taken actions based on the data assuming it to be secure.

Changing a Data Object from open to secure also risks information leakage. This is mainly because of the nature of AppBus Data Object references. The application doing the move would know of the change in secure status, but other applications may not. Since applications retain their reference to a Data Object after they acquire it, non-secure applications would then have a reference to a secure Data Object which other applications then would use as secure. This could lead to applications exposing secure data by modifying or updating this newly secure Data Object or taken actions based upon data they assume to be available only to trusted applications.

V. AppBus Usage and Applicability

Along with the discussion of the design and implementation of this technology, it is important to discuss why this approach is important. So, in that respect, it is useful to provide some usage examples. The first example is the same one given earlier in Section II.A, but this time with AppBus assistance. It shows several smaller, task-focused applications collaborating to provide a comprehensive travel planning application. Figure 2 is a graphical representation of these activities. The second example involves several communications applications exchanging data to make the user's experience more intuitive. After the examples, we talk about the complexity problem associated with multiple applications trying to exchange information.

A. Travel Planner

A user needs to make arrangements for an upcoming business trip using a mobile device. This will involve booking flights, reserving a hotel room and making a dinner reservation.

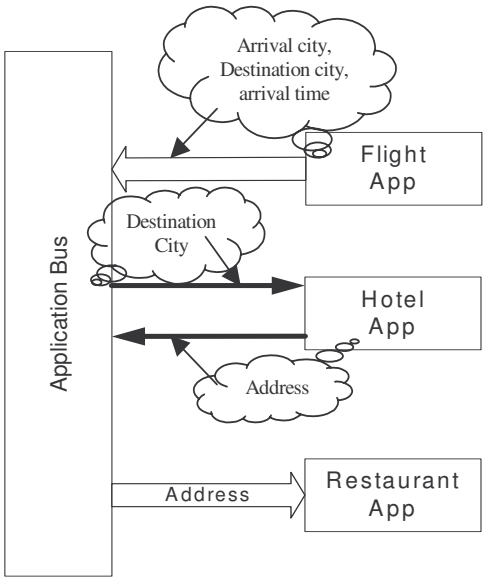

Figure 2. Travel Planner

- The user runs a flight-reservation application and books flights. This will involve entering some information, such as destination city and dates.
- All the details of his flight like time of arrival and departure, destination city, airlines become available on the AppBus.
- The user runs a hotel reservation application, which finds the destination city and travel dates available

on the AppBus. Sample reservations may be pre-populated allowing the user to simply choose one.

- The detailed hotel info becomes available to the AppBus, including the address of the hotel.
- The user now pulls up another application to reserve a table at the nearest Italian restaurant. This application gets the hotel address and arrival time from the AppBus to suggest a reservation time at a nearby restaurant, eliminating the need to reenter this data.

B. Communication Manager

The communication manager usage scenario shows implicit coordination among the various communication oriented applications on the device. The information exchanged is not necessarily task oriented, but shows how data tends to follow the user from one application to another.

- A user looks up a person in the Address Book. The Contact data becomes available on the AppBus.
- The phone application is started to make a call. The number available on the AppBus is displayed and the phone asks if this is the number to be dialed.
- Similarly, when the user switches to Instant Messaging, the IM address of the Contact on the AppBus is suggested. The user confirms this assumption and an IM session is launched.
- Similarly, if the user switches from IM to the Address Book, the IM application has placed data on the AppBus such that the other person's Contact data can be immediately displayed.

In these examples, the AppBus holds Contact data for the person recently interacted with or investigated, allowing applications to make educated guesses about the user's communication targets.

C. Application Communication Complexity

One problem that exists in complex systems is interdependencies. In order to participate in an application ecosystem, an application must know how to communicate with other applications or become increasingly less useful as the ecosystem grows. This was mentioned earlier, in the introduction of this article. And, as stated there, without some sort of infrastructure to help with the burden, an application author can make one of two choices: 1) explicitly integrate with other applications, via direct calls to their API's and queries on their data stores; or 2) push the burden back on the user. Obviously, placing more burden on the user is not the best approach, so the application must maintain the interfaces. The number of point to point connections between applications grows quite rapidly as the number of applications grows. This issue has become a significant problem in the mobile phone environment since 1) mobile phones now contain a great deal of functionality and 2) mobile phones lack the expansive user interface found on desktop computers that allow the user to handle this problem through the use of cut and paste type tools. Figure 3 graphically shows the number of application to application linkages required for only six applications to have full connectivity, which is 15 linkages for those six applications.

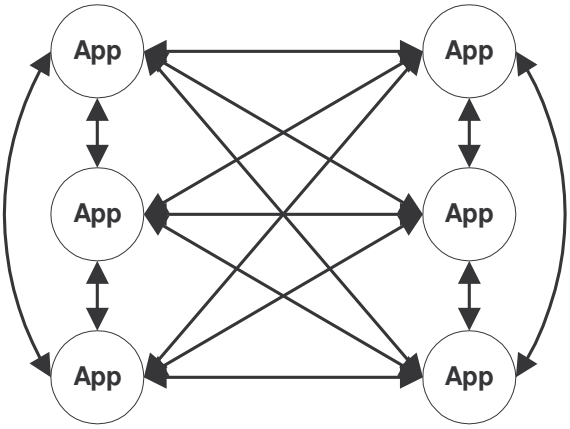

Figure 3. Direct Communication Links

So the burden on the individual application author becomes large for a given set of companion applications, but becomes larger still as we extend this over a period of time. As applications change, the companion applications must be aware of other applications and accommodate them accordingly.

On the other hand, with infrastructure such as the AppBus to act as the conduit between applications, the problem is much more manageable, as seen in Figure 4.

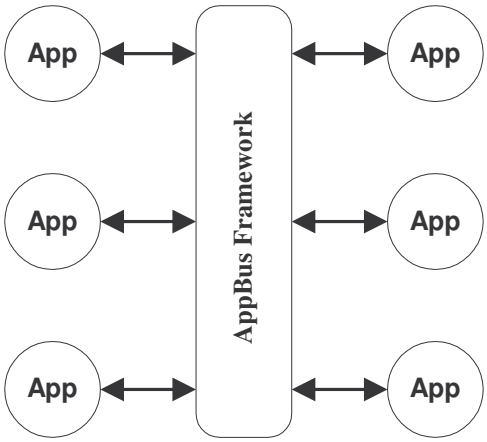

Figure 4. AppBus Assisted Communication Links

While applications still need to be able to understand data from other applications, the method of acquiring it and sharing it becomes much easier.

VI. Long Term Memory vs. Short Term Memory

There are a large number of methods and implementations of searching and indexing data available today. On the internet, search engines and indexes are common: Google[5][6], Yahoo[7], MSN[8] are a few. These engines look through existing data posted to the internet and create indexes. Then when a user requests information by means of a query, a (possibly) large list of

results is returned, accounting all sorts of data relating to the query's search terms. The result list is generally sorted by some sort of proprietary ranking system in an attempt to get the most relevant results to the position of prominence. For the most part, this system works well given the huge amount of data indexed.

There are similar implementations for searching on local computers as well. Google Desktop[6][9] is one example of this. This program creates an index of the data on the local machine and uses it to return results based on query parameters. Apple Spotlight[10] is another example of this technology for Apple's Mac OS X. A third example of this technology is the Beagle project[11]. This implementation runs under the Linux operating system.

The common thread through all of these implementations is that they search through a large number of long term data stores and create a large index in order to categorize the data processed. Data is retrieved through queries. The intelligence of understanding all of the different types of data stores lies in the framework.

The AppBus approach is different in several respects:

1) Short term data is indexed – While often a complete search of all stored data is what is required, there are times when current, contextual information is exactly what is needed. By keeping the data short term, application communication and collaboration can be enhanced by sticking to a set of the most relevant data items.

2) Intelligence identifying publishable data lies with the producer application – In AppBus, the framework does not know about application specific data stores. Consequently, it does not need to be updated with every new application. Additionally, the AppBus framework does not try and guess which items of data in an application's data store are most relevant. The application that created and is currently operating on the data is allowed to choose the most important pieces and publish them accordingly. While this creates a slightly higher burden on the application, it can also produce more relevant contextual results.

3) Notifications along with queries – The user or the applications do not need to actively query the framework in order to receive information. Notifications of new data and updates to existing data can be delivered to interested applications asynchronously. This allows an application to be passive until something of interest to it occurs.

VII. Comparative Technologies

There are already various technologies in existence that allow for data sharing between applications. None that we found fit the requirements we set forth. However, it is useful to look at existing ideas and implementations to discover what these other implementations included and how they were enabled.

A. Desktop Clipboard

The clipboard is a simple, user driven data sharing facility for desktop computers. For our goals, its valuable aspects include notification of clipboard changes and clipboard data inquiries.

However, the clipboard does not meet a number of our goals. Most important is the amount of explicit user interaction required to use the facility. Implicit data sharing is a primary goal of AppBus.

B. Tuple Spaces

A generic tuplespace shared memory exchange is a well-known method for communication between processes[12]. They have many qualities that fit our goals, and there are a number of implementations available[13][14]. One of these is the ability to exchange data between applications without either application knowing of the others existence. Tuples are added to the tuplespace and have their own identity at that point. Another valuable feature is data type flexibility. Tuplespaces are not limited by predefined schemas.

On the other hand, tuplespaces also have several features that make them unsuitable for our purposes. First, tuples are not modified once they have entered the tuplespace. For our purposes, it was important that the shared data be able to evolve as its usage changed. Another unsuitable feature is that tuples remain in the space until retrieved. In the AppBus, unused items eventually go away (via garbage collection).

C. Relational Database Management System

A Relational Database Management System (RDBMS) stores data in a database consisting of one or more tables of rows and columns. Along with many available implementations[15][16], there are many obvious desirable characteristics of an RDBMS. The ability to query for desired data and have event based updates are useful for our purposes. In fact, these characteristics are incorporated into the functionality set of AppBus.

Of course, an RDBMS also has a number of characteristics that makes it unsuitable for our purposes. The formality of establishing a schema is chief among them. This causes them to be not nearly flexible or ad-hoc enough for our purposes.

D. D-BUS

D-BUS[17][18] is a generic IPC mechanism that runs under Linux. The basic unit of IPC on D-BUS is a message, not a byte stream. D-BUS can be used as a standard IPC data passing mechanism similar to UNIX domain sockets, it can facilitate sending events through the system and it can act as an RPC mechanism allowing one application to request services from and invoke methods on another.

D-BUS does allow for asynchronous communication and somewhat generic message formats. However, it fails in meeting a number of our goals since it does not persist data and is not therefore queryable.

E. Desktop Search

As mentioned previously in section V, there are various implementations available today that index the contents of the data stores on a typical computer[9][10][11], such as document files, saved email and IM logs. These technologies typically create an index

on the local machine of words used in the corresponding data stores.

This technology is somewhat the opposite of the previous one, D-BUS, in that it persists data but is not made for asynchronous notifications. That point and the fact that the indexes can end up being rather large makes this unsuitable to meet many of our design goals.

VIII. Conclusions and Future Work

This paper describes the AppBus data sharing infrastructure. It differs from other data transfer technologies in that AppBus is designed to implement a more intuitive and collaborative method of current, contextual data sharing.

Enhancement and usage of AppBus is an ongoing project in our lab group. It has become a foundation technology in several of our projects to enable centralized application communication, event notification, uniform message passing and intuitive application cooperation. Several generational improvements are planned with the next already in the works. Among the issues currently being addressed are security (and its tradeoff with ease of collaboration) and the ontology and semantics of the data being shared.

Along with functionality improvements, we are coding more applications as data/event sources and sinks. Combining this with user studies should allow us to get a better idea of the quantitative benefits of our system.

References

[1] http://www.gsmarena.com/motorola_mpx-673.php
[2] http://www.gsmarena.com/motorola_e680i-1136.php
[3] http://www.gsmarena.com/motorola_a1200-1429.php
[4] L. Gong, "Java security: present and near future," IEEE Micro, vol. 17, issue 3, pp. 14-19, May 1997
[5] http://www.google.com/
[6] M. Cusumano, " Google: What It Is and What It Is Not," Communications of the ACM, vol. 48, issue 2, pp. 15-17, February 2005
[7] http://www.yahoo.com/
[8] http://www.msn.com/
[9] http://desktop.google.com/?promo=mp-gds-v1-1
[10] http://www.apple.com/macosx/features/spotlight/
[11] http://beagle-project.org/Main_Page
[12] D. Gelernter, "Generative Communication in Linda," TOPLAS 7, No. 1, 80-112 (1985).
[13] TSpaces, http://www.almaden.ibm.com/cs/TSpaces/
[14] JavaSpaces, http://developers.sun.com/
[15] Infinity Database, http://boilerbay.com/infinitydb/
[16] SmallSQL, http://smallsql.sourceforge.net/
[17] J. Palmieri, "Get on D-BUS", Red Hat Magazine, issue #3, January 2005
[18] http://www.freedesktop.org/wiki/Software/dbus
[19] C. Janssen, M. Pearce and S. Kollipara, “AppBus: Providing Short Term Memory for Mobile Devices”, CCNC 2006

Craig Janssen received a B.S in computer engineering from Iowa State University at Ames, Iowa. He is currently a Senior Staff Engineer at Motorola Labs, Motorola in Schaumburg, Illinois.

Michael Pearce received a B.S. in computer engineering from Iowa State University at Ames, Iowa. He is currently a Distinguished Member of the Technical Staff at Motorola Labs, Motorola in Schaumburg Illinois.

Nitya Narasimhan received a Ph.D. in computer engineering from University of California at Santa Barbara, California. She is currently a Principal Staff Engineer at Motorola Labs, Motorola in Schaumburg, Illinois.

Shriram Kollipara received an M.S. in electrical engineering and computer science from the University of Illinois at Chicago, Illinois. He is currently a Senior Software Engineer at Mobile Devices, Motorola in Libertyville, Illinois.