A Framework with Peer Fostering Mechanism for Mobile P2P Application Development

Heiko Wolf* and Mengqiu Wang†
Information Science Department
University of Otago
Dunedin, New Zealand
[*hwolf1, †mwang]@infoscience.otago.ac.nz

***Abstract*— The design of ad-hoc, wireless, peer-to-peer applications for small mobile devices raises a number of challenges for the developer, with object synchronisation, network failure and device limitations being the most significant. In this paper, we introduce a framework for peer-to-peer application development that deals with those problems. Other than most current literature, we focus on small peer-to-peer networks for gaming applications.**

I. Introduction

With small, mobile devices becoming more powerful, peer-to-peer applications on such devices are becoming increasingly popular. Full scalability, mobility and flexibility are desired features in many application domains, which can be achieved via mutual exchange of information and services over ad-hoc, wireless, peer-to-peer networks. Challenging problems arise in the development of such applications. Firstly, wireless, ad-hoc networks face problems such as stability, data integrity, routing, notification of joining and leaving peers, and, in case of peer failure, fault tolerance. In such networks, the connections of the devices may be highly variable as device may hop from online to offline unpredictably, and thus not reliable. Secondly, since the peers exist in a collaborative environment without central control, synchronisation of peers and distribution of resources become big issues. Thirdly, for the applications to run smoothly on small devices that we are aiming for, efficient management of local computing resources is a necessity. Those problems have been addressed in other research works, but mostly in an ad-hoc fashion [5], [12], [13].

When developing new applications, these issues have to be treated repeatedly as there exists no generic infrastructure which addresses all aforementioned problems. In this paper, we present a framework called "FRAGme2004" which we designed for developing collaborative mobile applications that achieve the features mentioned above by using a flexible peer-to-peer architecture.

The FRAGme2004 framework has a three-layer architecture. The layers inter-communicate via interfaces thus to achieve clear separation. The bottom layer is the Infrastructure Layer. This layer consists of the basic building blocks that address the communication requirements. A layer higher is the Object Layer. Object is the smallest entity that is distributed among the peers. The information and data that needs to be shared in the applications is encapsulated into objects, and this layer takes care of the delivery, synchronization and life-cycle management of objects. The top layer is the Application Layer. A clearly defined API is provided to the developers for easy application development.

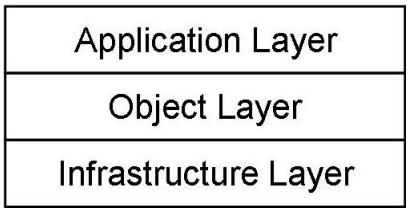


Fig. 1. Layers of FRAGme2004

The FRAGme2004 framework frees application developers from networking and resource management. This includes the establishment of the underlying network infrastructure, notification of joining and leaving peers, communication and object exchange. Rather, they can focus on the higher layers of the application. To make applications run reliably at all times, we build a "Peer Fostering" mechanism in the object layer, which makes use of data redundancy to achieve the overall integrity and robustness even in cases of node failure. The general framework and the "Peer Fostering" mechanism are the main contributions of this paper.

The paper is structured as follows: In Section II we will review related literature, followed by an introduction to the system architecture in Section III. Specific problems that arose and their solutions are introduced in Section IV, while Section V shows some mobile gaming applications developed with FRAGme2004.

II. Literature Review

With the take-off of vast mobile communication networks and technology, Peer-to-peer (P2P) computing has established itself as the next-generation distributed computing. A good introduction on P2P systems can be found in [14], which comprehensively reviews the field of P2P computing and applications. Besides giving a general overview, case studies of prominent peer-to-peer systems such as Groove, Gnutella and Freenet outline the major challenges in peer-to-peer networks. This review also shows the distinct differences between large-scale P2P applications such as file-sharing and

instant messaging, which have been the main research interest in many papers, and smaller scale P2P applications such as P2P gaming. FRAGme2004 is designed to facilitate the development of small-scale P2P applications that are typically hosted in a wireless ad-hoc network environment. Another good reference for an overlook of P2P computing is [15]. This book gives a step-by-step introduction from the background knowledge to the technical innovations of P2P computing. It also explores some technical issues associated with current P2P implementations. Particularly to our interest, the book mentions "remembering important information" as "a true sign of P2P intelligence"[15](p.145).

A topic discussed often is the one of data availability. [5] argues that "a complete system must ensure that important data remain preserved even if the creator or publisher leaves the system"(p.2). They developed a reliable preservation service built on a P2P architecture, on top of which digital library applications could be built. Another strategy, the "dissemination tree", is used in [4] to reduce the number of replicas and the bandwidth needed for updates. Finally, [12] introduces a protocol to handle the loss and rebuilt of replicas. All those approaches are aimed at applications such as file sharing, where data loss and node failure harm the performance, but not the overall functioning of the system. Data availability and reliability is one of the design goals of FRAGme2004. In particular, we aim at small scale P2P networks where every node failure can be a fatal threat to the functioning of the system. Our approach to this problem will be presented in more detail in Section IV-B.

Addressing a similar issue, [3] gave an interesting discussion on replicating mobile databases in peer-to-peer systems. In the collaborative applications that FRAGme2004 is targeting at, every individual peer will converge to a state where it contains the complete information in the system as shared among others, in order to collaborate. The concept that every peer has a copy of all objects is used in FRAGme2004 as intrinsic data replication to guard the overall data integrity and availability.

Many other important problems exist in P2P systems. [16] and [18] look at the problems caused by ineffective routing in ad-hoc P2P networks and how it can be improved. General problems of peer-to-peer collaboration such as collaboration over low bandwidth are discussed in [13]. They propose to mirror objects and translate changes to other peers in a network, an approach similar to the synchronisation protocol used in FRAGme2004. These problems are beyond the scope of FRAGme2004 and thus will not be covered in much detail in this paper.

FRAGme2004 is not the first effort to develope a framework for P2P applications. There have been different approaches in developing frameworks to allow the easy construction of peer-to-peer applications. [1] proposed a framework which includes group management and multicasting support to assist the design of complex applicactions. Another architecture that incorporates a variety of devices that communicate in a peer-to-peer network is introduced in [11]. While most P2P applications up to date can be narrowed to either file sharing or instant messaging categories, [7] introduces a framework that supports P2P services of any kind. Different from all these framworks, FRAGme2004 aims at small-scale P2P applications over wireless ad-hoc network. Therefore, we tackle some challenges related specifically to small-scale systems, which we found not well covered in previous researches.

III. The FRAGme2004 System

FRAGme2004 is a framework for the development of distributed, peer-to-peer applications. It offers a range of features that enable the easy development of reliable distributed applications. These features are introduced in the following sections.

A. Strict seperation of framework and application layer

FRAGme2004 takes care of all communication and resource management for application programmers. This allows the programmers to focus solely on their domain-specific problems. By providing an easy-to-use, stable and well-tested platform, FRAGme2004 helps to speed up the development of collaborative peer-to-peer applications. All framework functions are hidden from the application programmer and can be accessed through a clearly defined interface.

B. Reliable networking

Reliable networking is an important prerequisite for the usability of our framework. Several key points are identified to be crucial and briefly explained in the following paragraphs. Since we focused more on the construction of the higher layers of the framework in this project, we used a middleware called JGroups [10]. JGroups is a successful open source project for reliable group communication. It is very easy to use, offers all the key features and functionalities that we need and could be well integrated with the higher layers of FRAGme2004.

1) Group Management: The notion group is referred to as a number of peers engaged in a common activity or application. A group consists of one or more peers that can mutually share and exchange information; they can be addressed with a unique group identity while keeping their individual peer identity. Group management involves the creation and deletion of a group, joining and leaving a group as well as notification about joined/left/crashed members. Group members can be spread across LANs or WANs. The importance of group management is obvious. While group management appears as a simple concept, it conceals more low level details, for example the establishment of the network structure and communication channels.

2) Efficient Multicasting and Unicasting: Almost all peer-to-peer applications require efficient communication. This is especially true with complex applications like games. Unicasting - the point to point communication, albeit used in many places like file sharing applications, does not suffice here. In circumstances where peers in a group share the same resource or need to be notified at the same time for synchronization purposes, multicasting is called in.

For the FRAGme2004 framework we tested two approaches, one based on Remote Method Invocation (RMI) unicasting and one based on Multicasting, to determine the more efficient way of communication. It could be shown, that multicasting communication is significantly more efficient than RMI calls in group communications, an effect that aggrevates with an increasing number of peers. We attribute this to the fact that in case of RMI, a sender has to contact every other peer one by one in attempt to share information with his group; while in multicasting he only needs to make one sending for the information to be disseminated to all peers of his group.

3) Guarantee of delivery: Reliability is a big problem faced by wireless communication. Virtually no routing protocols can guarantee hundred percent success of delivery. But with some failure detection and retransmission mechanisms as included in JGroups, we can assume the guarantee of delivery to a certain degree. Such a guarantee is very important in terms of object-level protocol design. It allowed us to implement a high level protocol with less overhead and thus higher efficiency.

C. Efficient memory management

Having in mind that the applications need to run smoothly on memory-constrained small devices, we started addressing memory usage issues right from the beginning of the development of FRAGme2004. Software development patterns like Factory Pattern and Singleton Pattern [2] are used throughout the framework to ensure effective management of available memory.

D. Object synchronisation

The object synchronization process is taken care of by the framework internally and thus hidden from the application developer. All a programmer needs to do is to make a simple method invokation on the object that needs to be synchronized at the appropriate time point.

E. Recovery from peer failure

Peer failure, either transient or lasting, could cause serious problems in an application. This applies especially to games as usually the interaction of all players is needed to continue playing.

In the FRAGme2004 architecture, it is the application developer's task to implement the game logic after a peer dropout. However, the framework takes over the detection of leaving or failed peers and sends a notification to the application so this can be handled appropriately.

F. Usage Scenario

In this section, we will explain the main steps to develop FRAGme-based applications. We will also point out the interactions between the layers of our framework.

The FRAGme framework has a clear interface to develop FRAGme applications. The methods required by an application to interact with the framework and other peers are all included in one class called "ControlCenter". Application developers therefore import the ControlCenter class and use it to interact with the framework without the need to worry about any underlying code in the framework.

There is a number of steps that have to be pursued when developing a FRAGme application. First, we need to setup the connection, which is done by one simple method call. The underlying code in the object layer makes calls to the infrastructure layer to create the network connection and to notify other peers that a new peer has joined.

After the connection is setup, we need to receive all the FRAGme objects currently in the application, which is also done by invoking one method. The object layer takes care of gathering all objects from the other peers. After getting other objects, the new peer is setup and can now create its own objects.

Objects must be managed by the object layer in a unified fashion, hence it is important that objects are only created through the object layer manager. This is because the FRAGme framework uses the Factory Pattern in order to have absolute control over instantiating of relevant objects and to encapsulate object creation in one place. This design decision is made to allow full black-box development, no inside knowledge of the framework is required for application development. This also allows for sophisticated memory management, as the user cannot simply build instances of these objects, but must use the factory class.

When a so created object needs to be changed, the other peers must be notified. This is done in the FRAGme framework by invoking a simple change method on the object itself. When this method is called, it serializes the current object in the object layer and calls a method in the infrastructure layer to send it to the other peers. There the sent object is deserialized and updates the corresponding object on the remote side.

If a peer leaves the group intentionally or accidentally, the peer dropout mechanism comes into effect. As any peer's objects are fostered by another peer in the group, they are not lost. If the peer was holding any objects that all peers are using, such as the ball in a sports game, the ownership of these objects is transfered to the fostering peer. Any peer specific objects are just stored, but ownership is not transferred. In case the dropped out peer rejoins, these objects can be redistributed to him.

IV. The Specific Topics

During the development of FRAGme2004, a number of difficulties arose. They mainly come from one of the three areas: FRAGme2004 being a peer-to-peer system, challenges that come with wireless communication, and limitations of portable devices. We will describe how we cope with these problems in the following paragraphs.

A. Peer-to-peer related challenges

In a small peer-to-peer system, the main challenge is to achieve efficient object distribution and synchronisation. Especially in games, all peers have to be notified of object-changing events immediately to avoid stagnant gameplay.

In FRAGme2004, objects are assigned ownership, and by default objects are always owned by the peer that created them. To avoid cases where multiple peers invoking changes on the same object at the same time and cause possible race-condition or deadlock, only the owner of an object has the right to change the object. Every time a player interacts with an object that is not owned by him, the object's owner is requested to make the change. After the change has been made, the owner will then initiate a multicasting message to inform all other peers of the change.

B. Wireless communication related challenges

Wireless communication poses some additional challenges as compared to cable-bound communications. Reliability and packet loss are the major issues. At the low level, we rely on JGroups to provide us with reliable uni- and multicasting, but it doesn't save us from the problem at the higher level that applications suffer from a high degree of peer failure. Temporary disconnection of devices frequently occurs, especially in dynamic real-world environments.

This problem has serious impact on the basic usability of applications. In games for instance, it is generally not tolerable if a player is not able to continue the game just because of some temporary loss of connection (e.g., he walks into a lift). The problem becomes more serious if not only would that player suffer from the temporary disconnection, but also had other peers been affected. Therefore, it is important that some mechanisms are in place to ensure the integrity of data and continuity of the execution in case of peer failure. Furthurmore, it is also desired to have the disconncted peer being able to rejoin the application without loss of his previous data.

1) Introduction to "Peer Fostering" mechanism: To address the problems as per discussed above, we introduced a so-called "Peer Fostering" mechanism into FRAGme2004. Similar to the schemes used in some well-known peer-to-peer applications (Gnutella, Napster [15]), Peer Fostering is built on the basic idea that there exists some degree of data redundancy in all peer-to-peer systems. Most of the other systems don't have any special scheme that optimizes the degree of redundancy, but rather leave the highly redundant data in the system. This works for systems where peers are highly capable terminals like PCs. But in our case, all devices are very limited in terms of available memory. Therefore leaving such a high degree of redundancy could significantly constrain the performance of applications. As the number of peer failure instances increases, the accretion of redundant data could bring the system to crash.

We recognized that only a very small portion of the redundant data is required for the reliability of the system, Hence, we deviced a peer-fostering mechanism which builds dependancy relationships between peers.

2) Peer Fostering relationship : Having a set of existing peers in a system $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$ as the set of vertices in a graph, we use the symbol $Card(\mathcal{P})$ to denote the cardinality of set $\mathcal{P}$. If $Card(\mathcal{P}) > 1$, the Peer Fostering State (*PFS*) is defined as:

Definition 1: the peer fostering state of a system is a simple directed circle $\mathcal{C}_{pfs}$

$$\mathcal{C}_{pfs} = \{F(p_{i_1}, p_{i_2}), \ldots, F(p_{i_{n-1}}, p_{i_n}), F(p_{i_n}, p_{i_1})\}$$

where each occurance of $F(p_{i_a}, p_{i_b})$ is a directed edge in the circle from vertex p_{i_a} to vertex p_{i_b}. For any particular $\mathcal{C}_{pfs}$, there exists a bijection function that maps elements in the set $\{p_{i_1}, p_{i_2}, \ldots, p_{i_n}\}$ to the set $\{p_1, p_2, \ldots, p_n\}$.
Each edge $F(p_{i_a}, p_{i_b})$ also denotes a "fostering" relationship between peer p_{i_a} and peer p_{i_b}. In such a relationship, we say p_{i_a} is "fostering" p_{i_b}, with p_{i_a} being the "Foster" and p_{i_b} being the "Fostee". Since there is a one-to-one and onto function that makes the mapping between vertices in $\mathcal{C}_{pfs}$ and nodes in $\mathcal{P}$, each vertex p_{i_x} in $\mathcal{C}_{pfs}$ has an in-degree of 1 and an out-degree of 1. In other words, each peer p_{i_x} is acting as a "Foster" in one relationship and acting as a "Fostee" in another relationship.

When a peer drops out and the objects that he owns are not needed by the others, these objects still have to be sustained to allow rejoining. In our system, instead of having these objects taking up memory storage on every peer, only the foster of the drop out peer needs to store them. The "Foster"-"Fostee" relationship is illustrated in Figure 2.

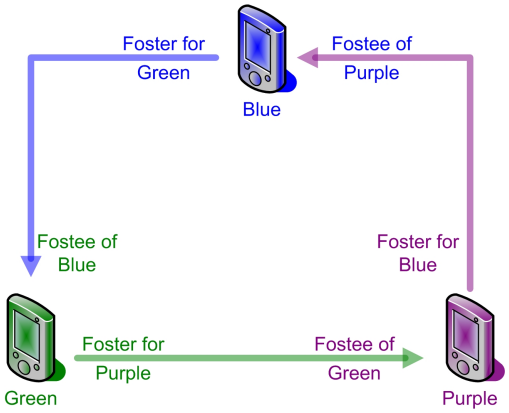


Fig. 2. Illustration of Foster-Fostee Relationship

In the "Peer Fostering" scheme, each participating peer not only knows who its "Fostee" is, but also knows whom its "Fostee" is fostering. The local knowledge of a peer can be expressed as:

Definition 2: the knowledge of peer p_i is

$$K(p_i) = \langle F(p_i, p_j), F(p_j, p_k), Adopt(\{p_{x_1}, \ldots, p_{x_m}\})\rangle$$

where the set $\{p_{x_1}, p_{x_2}, \ldots, p_{x_m}\}$ is the peers that p_i has "adopted", temporarily holding the objects of other peers.
The reason for storing this extra information will be clear when we explain the drop-out scenario in Section IV-B.3.

It is to be noticed that such relationships among peers are updated every time a new peer joins, or an existing peer drops out. We have a set of interaction protocols that allow the peers to make conversation and build the relationships. These conversations are initiated by the "active" peers - the

"Fosters". Although the reader will notice that every single peer is a "Foster" in some relationship, the peers don't have such global knowledge, and such knowledge is not needed. Each peer only makes sure that the relationship in which it is the "active" party is properly built. It is not hard to envision that when all peers finish building their own relationships, each peer will be fostered by some other peer.

This update of relationships happens in a separate thread behind the scene of the main gaming thread, and thread-safety measures are taken in our implementation. This relastionship building phase is essential, but it generates a very small amount of traffic to the network and therefore its impact is neglectable.

3) Peer drop-out handling: In this section, we will explain the sequence of actions that peers take in case of peer drop-out. We will use Figure 3 as illustration. In the initial configuration, the local knowledge of p_{purple} is:

$$K(p_{purple}) = \langle F(p_{purple}, p_{blue}), \\ F(p_{blue}, p_{green}), Adopt(\emptyset)\rangle$$

and the local knowledge of p_{green} is:

$$K(p_{green}) = \langle F(p_{green}, p_{purple}), \\ F(p_{purple}, p_{blue}), Adopt(\emptyset)\rangle$$

When a peer (p_{blue}) drops out, all other peers will be notified of the peer dropping event. p_{purple} will notice that p_{blue} matches the fostee in one of the relationships ($F(p_{purple}, p_{blue})$) that it knows locally. And because the foster in that relationship is p_{purple} itself, p_{purple} will first store all p_{blue}'s objects, updating the adopting set to be $Adopt(\{p_{blue}\})$, then notify other peers that the ownership of p_{blue}'s objects has been changed to p_{purple}. Knowing that p_{blue} was fostering p_{green}, p_{purple} will then take the initiative to reconstruct relationship by sending p_{green} a fostering request. Upon receiving such a request, p_{green} will send information about its own fostee, which in this case is p_{purple}. The local knowledge of p_{purple} will now be updated to be:

$$K(p_{purple}) = \langle F(p_{purple}, p_{green}), \\ F(p_{green}, p_{purple}), Adopt(\{p_{blue}\})\rangle$$

On the other hand, when p_{green} was notified that p_{blue} dropped out, it will notice that p_{blue} matches the fostee in one of the relationships ($F(p_{purple}, p_{blue})$) that it knows locally. And because p_{purple} is its current fostee, it foresees that p_{purple} will be fostering some other peer after the relationship has been reconstructed, and therefore it sends a request to p_{purple} to get the updated fostee of p_{purple}. After p_{purple} sends back the reply, the local knowledge of p_{green} will be updated to be:

$$K(p_{green}) = \langle F(p_{green}, p_{purple}), \\ F(p_{purple}, p_{green}), Adopt(\emptyset)\rangle$$

When p_{blue} rejoins, all peers will be notified of the joining event, and p_{purple} will notice that it is adopting p_{blue}'s previous objects. And if p_{blue} requests to have its previous data back (it has the alternative option to rejoin as a completely new peer), p_{purple} will send p_{blue}'s objects back, and transfer the ownership of these objects back to p_{blue}. This scheme can be scaled up to arbitrary number of peers. The peer fostering protocol is mentioned in greater detail in an internal technical report [17].

C. Device limitation related challenges

FRAGme2004 is aimed at easy and fast application development for small devices, such as PDAs or smart phones. Although the devices are becoming more powerful, memory and CPU constraints is still an important issue, especially when real time applications such as games are to be deployed on those devices. The device we used during the development of FRAGme2004 was the Linux-powered Sharp Zaurus SL-C700 [6], [19].

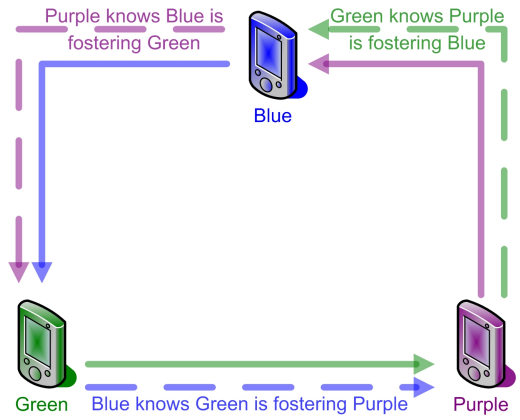
This PDA has a relatively powerful 400 MHz CPU which allows the display of complex graphics. The more important limitation was found to be the small memory available. After the Zaurus operating system was loaded, there were only 7 MB available for applications. This limitation called for a careful and memory efficient implementation of the object management. Also, the garbage collector of the Java Virtual Machine on the Zaurus did not work efficient enough which had to be considered as well.

Our memory management approach is based mainly on object reuse. As the garbage collection does not work efficient enough, the claimed memory of the application could grow very fast and crash the system if needed objects are created newly each time. Therefore, we have various mechanisms in place to reuse objects and thus keep the memory usage level low. Firstly, the Factory design pattern is used to control the creation of FRAGme objects. Once such an object is not needed anymore and disposed, it is stored and can be reused next time when a new object of the same type is demanded. Secondly, we employed Singleton design pattern so that all data stores that are used repeatedly, such as vectors that store peers or peer objects, are made static and global, making sure that they will only be created once.

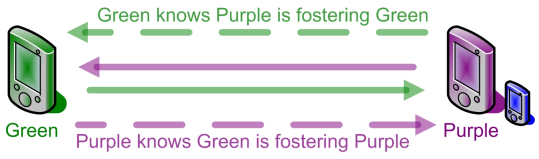
Although aimed for optimisation of running on the Zaurus, these design decisions will help to run FRAGme based applications on any Java powered devices with memory limitation.

V. The Games

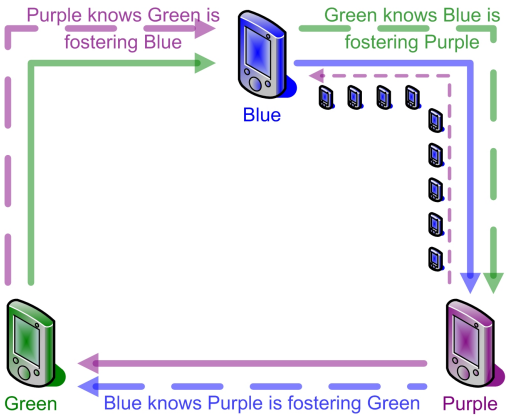
As a proof-of-concept, three networking-games that run on Zaurus were developed based on FRAGme2004 (a space shooter game called "SpaceBattle", a strategic tank game "BOOM!" and the Bomberman-like arcade game "RoboJoust"). A screenshot of RoboJoust can be seen in Figure 4. They show that memory and communication bandwidth constraints are handled well enough by FRAGme2004 to allow fast action games on a limited device such as the Sharp Zaurus. Also, minimal knowledge of the framework was required, which allowed novice developers to focus on the gameplay design. "BOOM" and "RoboJoust" were developed



(a) Initial configuration



(b) Blue drops out



(c) Blue rejoins

Fig. 3. Peer dropping out and rejoining

from scratch by a group of 8 fourth year students in under 50 hours.

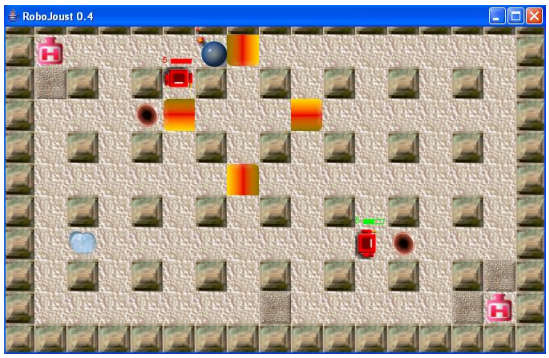


Fig. 4. Screenshot of RoboJoust

VI. Conclusion

As we showed in this paper, the development of mobile peer-to-peer applications poses a number of obstacles, with object synchronisation, network failure and device limitations being the most significant. With FRAGme2004, we developed a system that tackles those problems and offers a reliable framework for peer-to-peer application development. By seperating the application layer strictly from the framework infrastructure, FRAGme2004 allows developers to implement applications with minimal knowledge of the framework.

The impact of network failure and peer dropout is now efficiently reduced by our peer fostering mechanism.

For future development, the range of FRAGme2004 enabled devices can be further expanded. The development took place on the Sharp Zaurus SL-C700, which runs Java Personal Profile [9]. But not too many mobile devices support Java Personal Profile. To make FRAGme2004 more widely useable, it needs to be ported to the Java Mobile Information Device Profile (MIDP, [8]). Also, Zaurus devices communicate via WIFI. But since the network traffic generated by most FRAGme2004 applications is not unmanageable, it is possible to make FRAGme2004 incorporate other type of lower-capable wireless connections.

References

[1] Akehurst, D.H., Waters, A.G., and Derrick, J. (2004). “A Viewpoints Approach to Designing Group Based Applications”, In Herwig Unger, editor, *Design, Analysis and Simulation of Distributed Systems 2004*, Advanced Simulation Technologies Conference, pages 83-93, Arlington, Virginia, April 2004.

[2] Bruegge, B., and Dutoit, A.H. (2004). *Object-oriented software engineering: using UML, patterns, and Java*. Upper Saddle River, NJ, USA: Prentice Hall.

[3] Budiarto, Nishio, S., Tsukamoto, M. (2002). “Data management issues in mobile and peer-to-peer environments”, *Data & Knowledge Engineering*, Volume 41, Issue 2-3, pp. 183 - 204.

[4] Chen, Y., Katz, R. H., and Kubiatowicz, J. (2002). “Dynamic Replica Placement for Scalable Content Delivery”, In *International Workshop on Peer-to-Peer Systems*, March 2002.

[5] Cooper, B., Bawa, M., Daswani, N., Marti, S., and Garcia-Molina, H. (2003). “Authenticity and Availability in PIPE Networks”, *Future Generation of Computer Systems*.

[6] Device preview: Sharp Zaurus SL-C700 VGA resolution PDA, http://linuxdevices.com/articles/AT5295837592.html
[7] Gerke, J., Hausheer, D., Mischke, J., and Stiller, B. (2003). "An Architecture for a Service Oriented Peer-to-Peer System (SOPPS)", *Praxis der Informationsverarbeitung und Kommunikation (PIK)*, 2/03, p.90-95, April 2003
[8] Java MIDP, http://java.sun.com/products/midp/
[9] Java Personal Profile, http://java.sun.com/products/personalprofile/index.jsp
[10] JGroups, http://www.jgroups.org
[11] Kato, T. et al. (2003) "A platform and applications for mobile peer-to-peer communications", http://www.research.att.com/ rjana/Takeshi_Kato.pdf.
[12] Lin, S.-D., Lian, Q., Chen, M., and Zhang, Z. (2004). "A Practical Distributed Mutual Exclusion Protocol in Dynamic Peer-to-Peer Systems", *IPTPS04*.
[13] Margaritis, M., Fidas, C., Avouris, N., and Komis, V. (2003). "A Peer-To-Peer Architecture for Synchronous Collaboration over Low-Bandwidth Networks", in K. Margaritis, I Pitas (ed.) *Proc 9th PCI 2003*, Thessaloniki.
[14] Milojicic, D. S. et al.(2002). "Peer-to-peer computing", *Technical Report HPL-2002-57*, HP Lab, 2002.
[15] Moore, D., and Hebeler, J. (2002). *Peer-to-Peer: Building Secure, Scalable and Manageable Networks*. Berkeley, CA, USA: McGrawHill/Osborne.
[16] Naor, M., and Wieder, U. (2004). "Know thy Neighbor's Neighbor: Better Routing for Skip-Graphs and Small Worlds", In *The Third International Workshop on Peer-to-Peer Systems (IPTPS)*, 2004.
[17] Wang, M., and Wolf, H. (2005). "A Simple Peer Fostering Scheme for Reliable Peer-to-Peer Communication", *Technical Report*, Information Science Department, University of Otago, Dunedin.
[18] Ye, Z., Krishnamurthy, S.V., Tripathi, S.K. (2004). "A routing framework for providing robustness to node failures in mobile ad hoc networks", *Ad Hoc Networks*, 2 (2004), pp. 87 - 107.
[19] Zaurus SL-C700 Frequently Asked Questions, http://www.dynamism.com/zaurus/faq.html