LAPPEENRANTA UNIVERSITY OF TECHNOLOGY
Faculty of Information Management
Department of Information Technology

Bachelor's thesis

PROGRAM HOSTING IN COMMUNICATIONS ENVIRONMENT WITH REACTOR AND THREAD POOL APPROACH

Kimmo Kolehmainen
0237762
Department of Information Technology
kimmo.kolehmainen@lut.fi
Communications software

Supervisor: Petri Heinilä

ABSTRACT

Lappeenranta University of Technology
Faculty of Technology Management
Department of Information Technology

Kimmo Kolehmainen

Program hosting in communications environment with Reactor and thread pool approach

Bachelor's thesis

2009

49 pages, 14 figures, 1 table and 1 appendix

Examiner: Assistant Petri Heinilä

Keywords: Event-driven architecture, Reactor design pattern, thread pool, Leader/Followers design patter, PeerHood

In this thesis concurrent communication event handling is implemented using thread pool approach. Concurrent events are handled with a Reactor design pattern and multithreading is implemented using a Leader/Followers design pattern. Main focus is to evaluate behaviour of implemented model by different numbers of concurrent connections and amount of used threads. Furthermore, model feasibility in a PeerHood middleware is evaluated.

Implemented model is evaluated with created test environment which enables concurrent message sending from multiple connections to the system under test. Messages round trip times are measured in the tester application. In the evaluation processing delay into system is simulated and influence of delay to the average round trip time is analysed.

TIIVISTELMÄ

Lappeenrannan teknillinen yliopisto
Teknistaloudellinen tiedekunta
Tietotekniikan osasto

Kimmo Kolehmainen

Ohjelman isännöinti viestinvälitysympäristössä hyödyntäen Reactor-suunnittelumallia yhdessä säiealtaan kanssa

Kandidaatintyö

2009

49 sivua, 14 kuvaa, 1 taulukko ja 1 liite

Tarkastaja: Assistentti Petri Heinilä

Hakusanat: Tapahtumalähtöinen arkkitehtuuri, Reactor-suunnittelumalli, säieallas, Leader/Followers-suunnittelumalli, PeerHood

Tässä kandidaatintyössä toteutetaan yhdenaikaisten tiedonvälitystapahtumien hallinta hyväksi käyttäen moniajoa säieallas mallia hyväksikäyttäen. Työssä hyödynnetään Reactor-suunnittelumallia tapahtumien vastaanottamisessa, sekä sovelletaan Leader/Followers-suunnittelumallia toteuttamaan käsiteltävien tapahtumien moniajo. Keskeisinä tavoitteina on arvioida toteutetun mallin käyttäytymistä eri yhteyksien ja säikeiden määrillä, sekä toteuttaa ja arvioida mallin soveltuvuutta PeerHood välikerroksessa.

Toteutettua mallia on arvioitu luodulla testiympäristöllä, jonka avulla testattavaan järjestelmään voidaan lähettää viestejä yhdenaikaisesti monista eri yhteyksistä. Viestien kiertoajat ovat mitattu testiohjelmassa. Arvioinnissa on myös simuloitu järjestelmässä aiheutuvaa viivettä ja sen vaikutus lähetettyjen viestien keskimääräiseen kiertoaikaan.

TABLE OF CONTENTS

ABBREVIATIONS

CPU	Central Processing Unit
GPRS	General Packet Radio Service
I/O	Input/Output
POSIX	Portable Operating System Interface
RAM	Read Access Memory
TCP	Transmission Control Protocol
WLAN	Wireless Local Area Network

1. INTRODUCTION

Design of concurrent communication system has many challenges and developing communication concurrency is very error prone because implementation is usually done in the low level of the system. In addition, events can be arrived simultaneously from multiple event sources which make debugging difficult. Besides, used system architecture can have dramatic impact of total system performance. With using well-known design patterns common pitfalls can be avoided. In this thesis the Reactor design pattern [1] is utilized with a common thread pool solution.

Basically concurrent communication solutions are divided in two different groups. These groups are event-driven architecture and multithreading architecture [2]. This thesis study hybrid solution using the Reactor design pattern with a thread pool approach. Feasibility of solution is evaluated in low capacity devices like mobile devices point of view.

This thesis is divided into three parts, first the basic Reactor design pattern and two possible thread pool solutions are described and explained. After theoretical study the Reactor with selected thread pool approach are implemented first as a reference implementation to the POSIX environment and second to the PeerHood middleware [3] environment. A PeerHood implementation is done by replacing old concurrent connection handling to the Reactor with thread pool solution. Finally, solutions are benchmarked and evaluated with different system characteristics.

1.1 Objectives

Motivation of this study is to find more feasible concurrent communication solution to the PeerHood environment with better flexibility and limited resource usage. Purpose of this thesis is to implement the Reactor with thread pool solution to the PeerHood middleware and evaluate how feasible implemented solution is for the PeerHood. Second target is to provide reference implementation of the Reactor with thread pool event handling model.

2. DESIGN

Following chapters will give a brief description what is purpose of the Reactor design pattern and structure of the Reactor model is explained. Furthermore, some drawbacks of it are raised up. Finally, two possible thread pool models are presented and compared which one is more feasible in the scope of this study.

2.1 Reactor Design Pattern

The Reactor design pattern is an event handling design pattern [1] which is used to synchronous demultiplexing for concurrent events and dispatching those to specified event handlers [4]. Events – which can arrive concurrently from multiple sources – are demultiplexed typically using the POSIX *select()* function [5].

Reduced coupling between real event demultiplexing functionality and the event handling is provided with the Reactor design pattern. Reduced coupling improves system flexibility [1]. Because of an abstraction in the Reactor pattern it handles each event handlers in a same way. New event handlers can be added and existing can be changed without any modification in the Reactor implementation [4].

2.1.1 Structure of the Reactor

The Reactor design pattern is based on five main components [1]. In the Figure 1 is shown these components and their relationships. *Reactor* class acts in the main role of this design pattern where event handlers are registered as a listeners. When the *Reactor* object receives event from I/O-demultiplexing mechanism provided by operating system or application framework, it dispatches that to particular event handler identified by I/O handle [5].

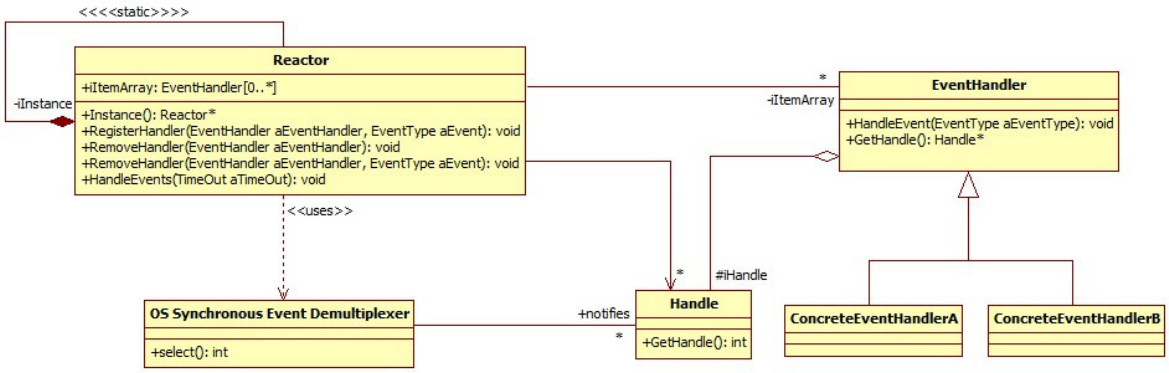

Figure 1: The Reactor design pattern, reproduced from [1]

Reactor class provides interface for register and remove *EventHandler* objects. As well there is ability to register several *EventTypes* to the same *EventHandler.* In the model shown in the Figure 1 is *Reactor* class implemented as Singleton design pattern [6] since typically only one Reactor is needed at a time [1]. *HandleEvents()* is used to activate the Reactor to receive events. If timeout value is set to zero *HandleEvents()* method will be blocked until some I/O-event occurs.

EventHandler class is used to abstract concrete event handlers which provide better system flexibility by using decoupling. *EventHandler* class defines method for receiving events from the *Reactor* object and accessory method for the *Reactor* to get *EventHandler* specific I/O-*Handle* [1].

With *select()* function three various event types can occur; read, write and exception event [5]. ***EventType*** enumeration specifies events which *EventHandlers* can use when registering to the Reactor. This enumeration is one parameter of *Reactor*'s class *RegisterHandler()* method. *RegisterHandler()* can be called several times for setting multiple events where *EventHandler* are interested in. When the Reactor notify occurred event it gives event type as a parameter of *HandleEvent* method. The Reactor design pattern can be extended to have timeout events and catch also POSIX's signals [5] which are then notified to registered handlers. [1]

Event handling callback method can be defined as well so that each event type has own callback method. Although that solution is not as flexible as delivering occurred event as a parameter. In that adding new event to system will cause change to defined interface. [1, 4]

Handle is a wrapper class [6] which wraps real platform specific I/O-handle to it inside. Wrapping gives higher portability than using system level I/O-handles directly in the system wide. Wrapped handle is uniquely identifier such an opened file or network connection descriptor. The Reactor use handles for I/O-demultiplexing and identify correct event handler for the occurred event [1].

ConcreateEventHandler class implements methods derived from *EventHandler* class. After registration to the Reactor this class waits events from the Reactor binded to handle which *GetHandle()* method returns. When event – which *ConcreteEventHandler* is waiting – occurs implemented *HandleEvent()* method is called. [1]

OS synchronous event demultiplexer is a system specific event demultiplexing functionality. In POSIX compliant systems *select()* function provides needed functionality [5]. With *select()* function *Reactor* class wait registered *EventHandlers* handles to occur. *select()* function does not return until at least one event is occurred or defined timeout is expired. *Select()* function can also return in error situation [5].

2.1.2 Components Collaboration

When *Reactor* class object is created event handlers can register events to the Reactor. Registered event handler contains handle which is real event source for the Reactor used in the *select()* demultiplexing function. Figure 2 shows messages between components in the registration and event notification.

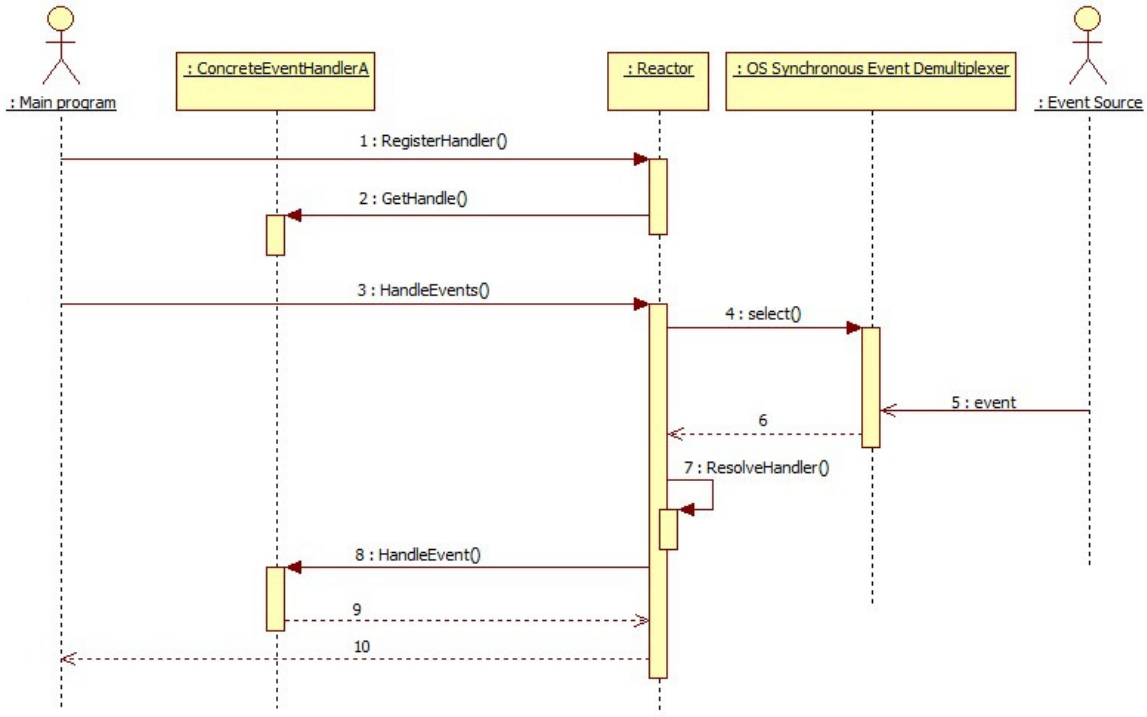

Figure 2: Collaboration between Reactor components, reproduce from [1]

Performance bottleneck which can affect the system response time and latency can be seen from Figure 2 [7]. After *select()* method returns *HandleEvents* method continue to resolve correct event handler and call it *HandleEvent* callback method. Events cannot receive during this time, hence these operations have to be done fast. Next event can be handled after *select()* function is called again. This performance issue can be significant especially in single thread applications. Impact of that problem can be decreased by using multithreading mechanism with the Reactor design pattern. [1]

2.1.3 Reactor in Multithread Environment

When using the Reactor design pattern in a multithreading environment there are some things what need to be take care. In multithread environment several event handlers can be registered or unregistered concurrently in the Reactor. Data corruption can be avoided using data synchronization when handling internal data in the Reactor [1]. Second issue is delay when register or unregister *EventHandler* to the Reactor. *EventHandlers* are not included to the used handle set until *select()* function returns and update used handles. That delay can be

avoided using some notifier handle which is registered to the handle set as well. With this notifier handle *select()* function can be notified about internal state change. [1]

2.1.4 Pros and Cons

The Reactor design pattern provides ability to create flexible and effective event-driven architecture for handling concurrent events. Event handlers do not need to poll events perpetually, which waste unnecessarily CPU time [5]. In addition, the Reactor design pattern scales up portability to the different environment. [1]

The Reactor model has some weakness as well. From Figure 2 can be seen that if *EventHandler*'s *HandleEvent()* method execution takes a lot of time the Reactor cannot receive new events. Long event handling time increases system response latency and decrease event throughput [7]. For feasible performance *select()* function must call fast again after it has return.

2.2 Thread Pool

Thread can be thought as a single execution unit, which operating system schedules to run. Threads in a same process can have access to same memory and descriptors like file and network handles. With threads application can perform several tasks in simultaneously. [5]

With multithreading, communication system latency and response time can be decreased. Although, data consistent must be guaranteed, this will lead more complex system design. [1, 5, 8]

There are several architecture structures to develop multithreading communication event handling. Three common structures are:

- **Thread-per-request** model creates a new thread for each request from the client. After creation, request is handled in the own thread [8].
- **Thread-per-connection** model creates a new thread for each new connection from the client to the system. After thread creation, every request for the connection is handled in created thread. [8]

- **Thread pool** is model which contains number of threads created as workers. These worker threads handle client requests. In the thread pool model there is no need to create each time new thread. Because of creating new thread is not needed it is more efficient than model where new thread is created when needed. In reason of creating new thread is not efficient and cause additional overhead. [8]

Compared as other models expensive thread creation overhead can be avoided in the thread pool model [9]. For that why system can be more efficient as long as there are threads left in the thread pool. If any thread is not available, requests need to wait until some thread completes its task. In addition, event handling memory consumption can be managed by size of thread pool. However, amount of threads will affect to the system response time. [1, 8, 9]

Although multithreading is more complex than single thread application it can improve a lot of application performance, like response time and throughput [5]. Using well-know solutions application programmer can avoid several multithreading pitfalls, like deadlocks and data corruptions [1]. In the next two different general solutions for the event handling with the thread pool approach are presented.

2.2.1 Half-Sync/Half-Reactive Design Pattern

A Half-Sync/Half-Reactive design pattern is thread pool based variant of the Half-Sync/Half-Async design pattern which is design pattern for decouple asynchronous events to synchronous service processing in the concurrent system [1, 10]. The Half-Sync/Half-Async design pattern consist three layers which are shown in the Figure 3.

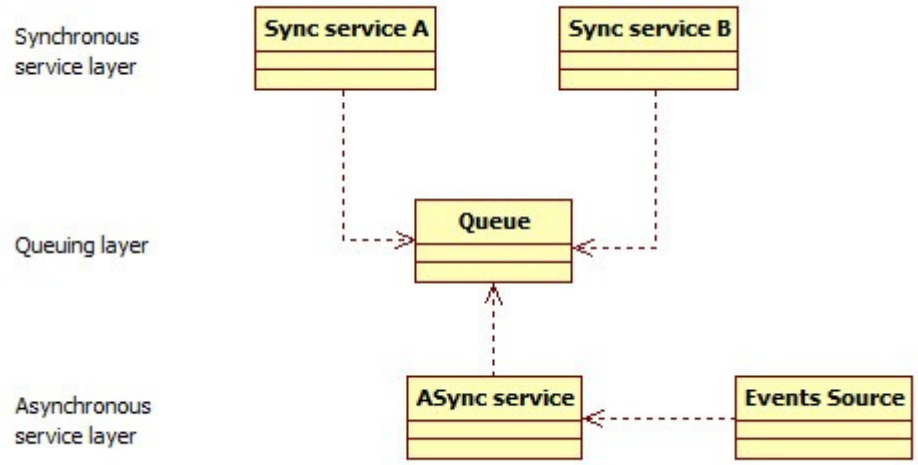

Figure 3: Half-Sync/Half-Async structure, reproduced from [1]

- **Asynchronous service layer** thread receives asynchronous event notifications from handle set. After event is received notification layer resolve correct event handle and reads transaction request from handle and allocate it dynamically to the queue. Asynchronous service layer is low level layer and event processing in this layer cannot take a long time [11].

- **Queuing layer** buffers input from asynchronous service layer to the queue and notify synchronous service about the input. Queuing layer is a mediator [6] which separates asynchronous service layer from synchronous service layer.

- **Synchronous service layer** receives input notification from queuing layer and handle input from the queue. Synchronous layer is high level layer where is long duration processing done.

More complex implementations can have several queues which can be used as prioritized queues for offering better quality of service. [1, 10]

The Half-Sync/Half-Reactive design pattern variation contains same layers as the Half-Sync/Half-Async design pattern. Furthermore, the Half-Sync/Half-Reactive use I/O-demultiplexing in the asynchronous layer and worker threads from thread pool to process messages from queue [1, 11].

The Half-Sync/Half-Reactive pattern benefits are ability to use high level synchronous programming, which can simplify service implementation. In addition, the Half-Sync/Half-Reactive model support request buffering and if needed it is easy to extend using thread borrowing [10]. However it has several drawbacks as well. Queue layer produce additional context switching and data synchronization. As well queuing causes additional data allocation and data overheading. [1, 10]

2.2.2 Leader/Followers Design Pattern

The Leader/Followers design pattern provides efficient concurrent thread pool strategy where multiple threads wait turn to demultiplex events [1]. The Leader/Followers can be more efficient than the Half-Sync/Half-Reactor in that there is no need to allocate new memory for every request and copy it somewhere as in the Half-sync/Half-Reactor pattern [12]. Furthermore, the Leader/Followers do not require as much data synchronization as using the Half-Sync/Half-Reactive [10, 11]. Table 1 show implementation differences between the Half-Sync/Half-Async and the Leader/Followers [12].

Operation	**Times called Half-Sync/ Half-Async**	**Times called Leader/ Followers**
malloc	2	0
free	2	0
enqueue	2	0
dequeue	2	0
signal	2	1
locks	8	2

Table 1 : Some operations in the Half-Sync/ Half-Async and the Leader/Followers implementation [12]

The Leader/Followers design pattern allows threads to coordinate themselves and protect critical sections. In the model one thread from thread pool is the leader at a time [1]. The leader waits until some event occurs from active handle set. After event has occurred the leader promotes one follower thread to the new leader thread and start acting as a processing thread for the occurred event. When event processing has finished thread returns back as the follower to wait turn to become the leader again. Figure 4 represent complete state chart of the Leader/Followers design pattern. [11]

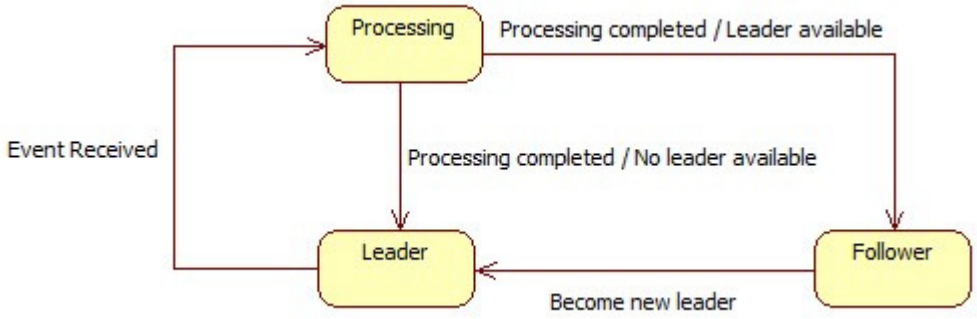

Figure 4: States of the Leader/Followers design pattern, reproduced from [11]

This kind of model can be thought as a thread-per-request model except used thread is taken from the pool if there is a free thread left [9]. In the pure thread-per-request model new thread is created on every request. Thread creation overhead can be avoided during request handling by using already created threads [8].

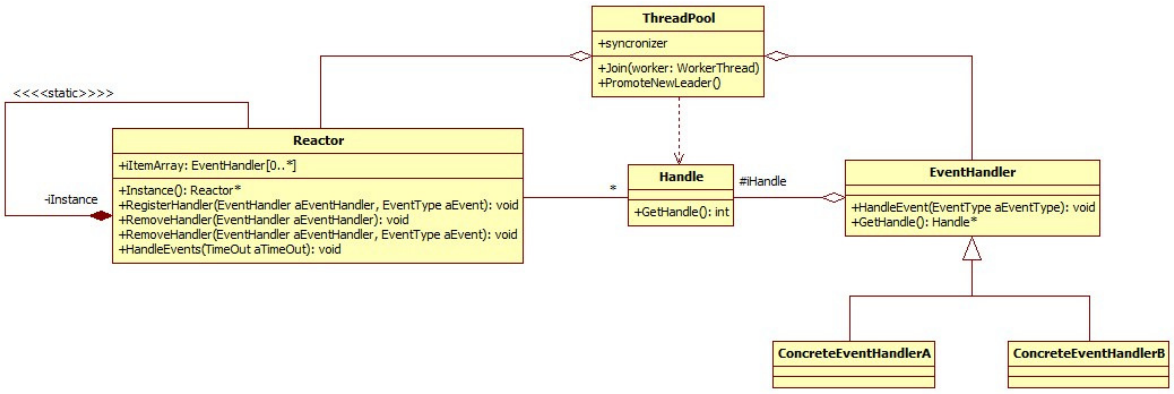

Figure 5: The Leader/Followers design pattern with the Reactor design pattern, reproduced from [11]

In the Figure 5 the Leader/Followers design pattern is presented with the Reactor pattern. Difference from the basic Reactor design pattern is a new *ThreadPool* component which is like heart of model. The *ThreadPool* contains worker threads which one by one wait events and processing those. [11]

2.2.2.1 Collaboration

In Figure 6 is shown collaboration between components in the Leader/Followers design pattern. Collaboration is done with the Reactor design pattern, which the current leader uses for demultiplexing events. First *WorkerThread* object join to the *ThreadPool* and because of thread pool has no leader *WorkerThread* A is promoted to the leader thread. When *WorkerThread* B joins to *ThreadPool* *WorkerThread* A is the leader so B become to a follower thread.

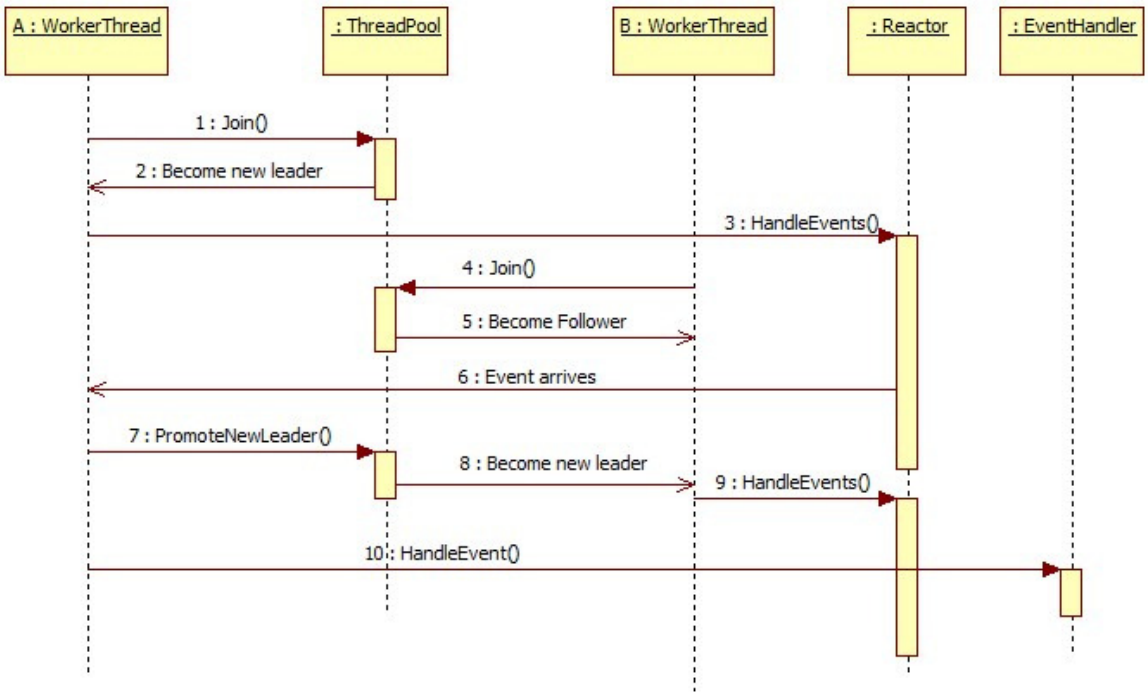

Figure 6: Collaboration of the Leader/Followers design pattern, reproduced from [11]

When the Reactor receive event, the *WorkerThread* A request from the *ThreadPool* to promote a new leader and A will continue processing received event. Meanwhile *WorkerThread* B has become the new leader thread and it has started to wait events from the Reactor. [11]

In the Reactor with the Leader/Followers model is important to remember deactivate handle which was caused occurred event. If this is not done, promoted thread can assume new event to be arrived, which will cause race condition between threads to handle the same event. For this reason handle need to be removed temporarily from active handle set before new thread is promoted. After *HandleEvent()* method is return handle can be set back to active handles. [1]

The Leader/Followers design pattern provides solution for longer event handling time, which was problem in the single thread Reactor model. If there is thread available in the thread pool with the Leader/Followers model events are started to wait again before occurred event is started to handle.

3. IMPLEMENTATION

The Reactor design pattern with the Leader/Followers pattern was first implemented as a reference implementation in the POSIX environment. Implementation was done using C++ programming language. After reference implementation was done it was utilized in the PeerHood, which is a peer-to-peer communication middleware [3]. Purpose of replacing old communication event handling to the Reactor with thread pool based approach was get more flexible solutions with lower resource usage than preceding solution.

The Leader/Followers thread pool pattern was selected because it simplicity and it appears more lightweight solution than the Half-Sync/Half-Reactive pattern. The PeerHood middleware is communication middleware for the mobile environment where resources are quite limited.

Reason for change preceding communication event handling in the PeerHood was to reduce usage of the system resource and find more flexible way to handle concurrent event handling from the concurrent event sources. With the new solution, system resource usage can be easily controlled by modifying numbers of threads in the thread pool. As well new event handlers can be added without any large changes.

3.1 Reference Implementation

Idea of reference implementation was to implement independent C++ application which uses POSIX compliant system resources. With reference implementation the Reactor with Leader/Followers pattern is introduced how it can be implemented. Furthermore, performance measurements for the model with different characteristics can be done effectively without additional disturbing factor. The reference implementation can be found in the Appendix 1: Reactor with Leader/Followers thread pool.

3.2 PeerHood

The PeerHood is a communication middleware for the peer-to-peer communication with a device neighborhood. The PeerHood concept is for mobile devices to monitor constantly services from other devices in the neighborhood. It supports functionality like:

- Detect other devices using different network technologies
- Discover services from other devices
- Advertise own services to other devices
- Monitor status of devices into neighborhood

Besides, the PeerHood middleware enable roaming between different wireless technologies. [3]

The PeerHood middleware is currently implemented on a Maemo platform which is based on the Linux operating system kernel and target to mobile devices such as Nokia Internet tablets [13]. With plug-in implementation the PeerHood support several network technologies like Bluetooth, WLAN and GPRS connections. In Figure 7 is shown basic concept of the PeerHood. [3]

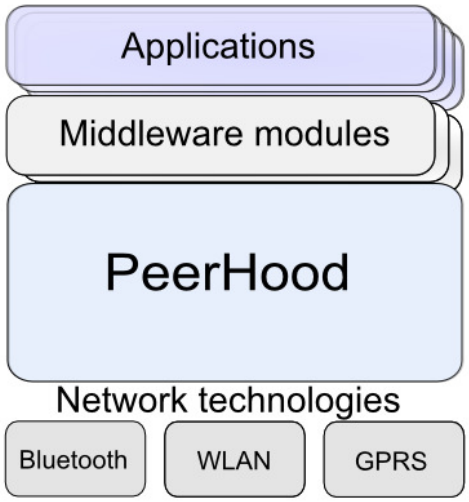

Figure 7: The concept of the PeerHood, reproduced from [3]

3.2.1 Structure of the PeerHood

The PeerHood system consist a Daemon process and applications which can be act as a service or be a client of a service from neighborhood. The Daemon process is response for collecting information of devices and services from the neighborhood. The Daemon also publishes information about itself. Multiple PeerHood applications and services can be run in same device and for that reason common functionalities are done in the one Daemon process. Applications use the Daemon indirectly through a PeerHood library (Figure 8). In addition, the PeerHood library delivers established connections to the application. After connection is

moved to the application, handling of connection depends entirely an application developer how he is implement the communication handling. [3]

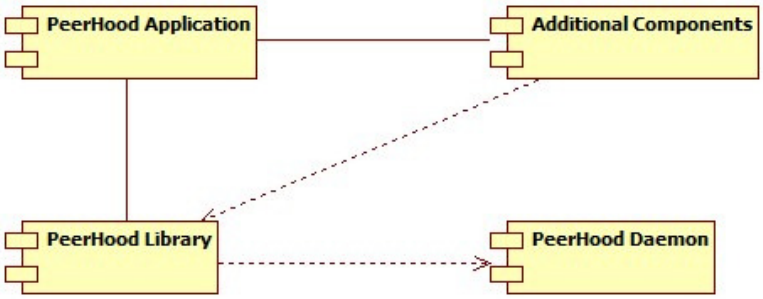

Figure 8: Component model of the PeerHood, reproduced from [3]

3.3 Utilization of the Reactor with the Leader/Followers in the PeerHood

After reference implementation was made implementing the Reactor with the Leader/Followers model to the PeerHood was very straightforward. Focus was replace connection establish and message handling between the PeerHood applications. Functionalities in the PeerHood Daemon were leaved out of scope.

The PeerHood library accept and establish connections with *select()* demultiplexing function. When connections are established it first removes connection from the handle set and after that, move connections to the application using callback interface. Common way was use thread per connection model after connection has moved to the application. Changed implementation publish access to a PeerHood interface for register and removes event handlers in the added Reactor. With the published methods application developer can add received connections to the Reactor and implement only event handlers (Figure 9). Decision of register connection to the Reactor is wanted to keep in the application developer.

Preceding connection accepting was moved to the acceptor event handler [1] which accept incoming socket and create new event handler, which is response for establishing connection as a PeerHood protocol defines it (Figure 9). After connection establishing is done without errors, event handler removes itself from the Reactor and transfer connection to the application and finally destroys itself.

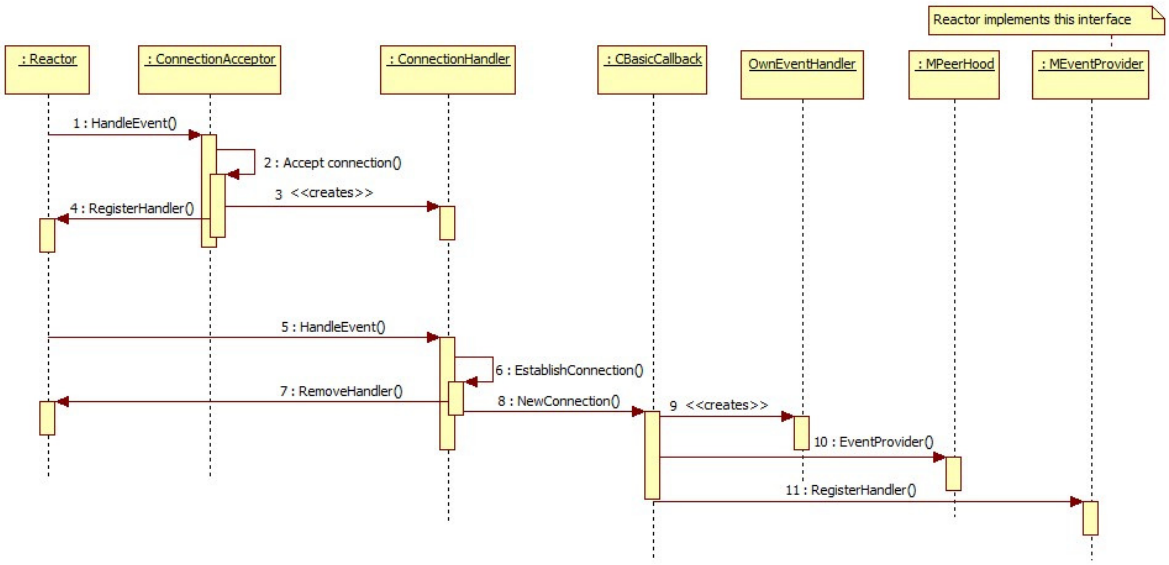

Figure 9: Implemented connection accepting and establishing

3.4 Future Considerations

Further study related to the Reactor implementation in the PeerHood could be figure some reasonable timeout event handling for the Reactor, so that event handlers could register timeout time to be notified. With timeout functionality the Reactor with Leader/Followers model can be implemented as well handling device discovery, monitoring and inquiring without creating own thread for each action in specific communication technology.

Dynamic thread creation and destroying for threads in the thread pool can reduce system resource usage without notable performance decrease. In that model all threads are not created at a same time in a startup. In the startup only few threads are created and others only when amount of concurrent connections increase over some specific level. Furthermore, some threads are destroyed if they are been long time without any activity. [14]

4. EVALUATION OF SOLUTION

Evaluation of implemented solution is considerate to performance testing for reference implementation and comparing preceded and the new PeerHood implementations together. Feasibility of implemented solution is examined about a low resource device point of view. Focus for evaluating is to bring up how different characteristics affect to the implemented model.

4.1 Benchmark Tests and Results

Performance of reference implementation is tested for evaluating independently the Reactor with Leader/Followers thread pool model. Purpose of measurements is to give overview how thread pool affect to the Reactor pattern performance and what kind influence of different amount of threads is to the model performance.

Benchmarking is done using message round trip time from a tester application to the system under test and back to the tester application. With this setup system total message handling time can be compared and evaluate how different characteristics will affect to the system. Results of the measures are mean values of round trip times. In all tests variance of times was increased according as number of concurrent connections increased.

The PeerHood implementation is benchmarked with earlier implementation. First test set is ran for the old implementation and results of it are then compared to the results of new implementation. Target for the Reactor with Leader/Followers implementation is to provide flexible solution which will decrease resource usage – like threads – without decreasing communication performance dramatically.

4.1.1 Test Bed

Basically both, the reference implementation and the PeerHood performance are measured in the same way. A proposed test bed is to use tester application, which creates number of simultaneous local TCP connections to the system under test. When connections are created each connection send sequentially ten times “Hello World!” -message to the system under

test. Connections runs in own threads so messages are simulated to send simultaneously from different connections to the system under test. The tested system receives messages and return received messages back to the tester application. The tester application measures times between messages sent and received (Figure 10). The tester application measures times between point 1 and point 5 from Figure 10.

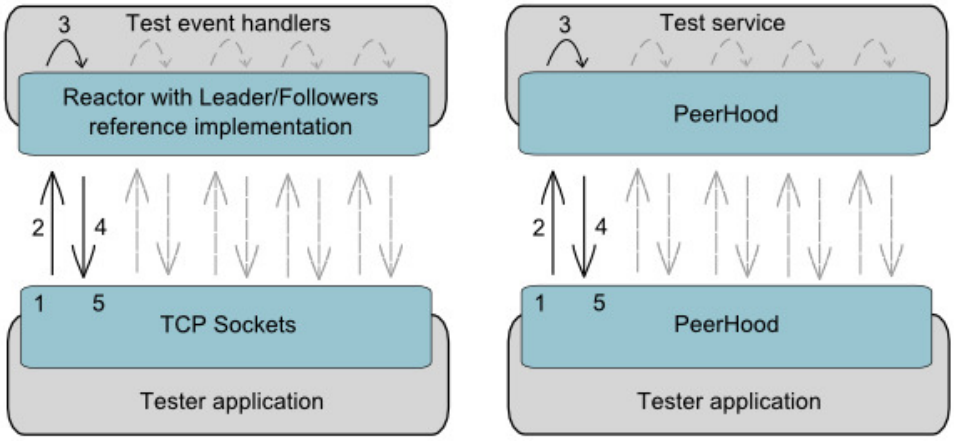

Figure 10: Benchmark setups for the Reactor with Leader/Followers (left) and the PeerHood (right) implementations

Tha proposed test bed as well includes ability to simulate processing delay before received message is send back to the tester application. In the point 3 processing simulation is done with looped sleep, which runs micro sleep into while loop for given time (Listing 1).

```
int i = 0;
while( i < processingTime )
    {
    usleep( 1 );
    i++;
    }
```

Listing 1: Processing time simulating

4.1.2 Used Hardware and System Environment

Reference implementation and PeerHood implementations were measured in a Ubuntu 8.10 (intrepid) operating system with 2.6.27-11-generic version of the Kernel. Used hardware was Compaq Evo N1020v laptop consisting 723.4 MiB RAM memory and 1.6 GHz Mobile Intel Celeron processor. Furthermore, in PeerHood implementations tests were ran into Maemo

4.1.2 Diablo x86 environment, which uses a Scratchbox Apophis R4 cross-compiling environment [15, 16].

4.1.3 Results of the Reference Implementation Measurements

The reference implementation behavior tested with different thread amounts and with and without simulated processing delay. In all cases “Hello world!” -message was sent to the tested system. Round trip times of the messages were measured in the tester application.

In Figure 11 are shown measured results without simulated delay. Results show that in a small work load differences between used threads are very slight in a small amount of connections. Difference between used amounts of threads can be seen after concurrent connections have increased over the 64 connections.

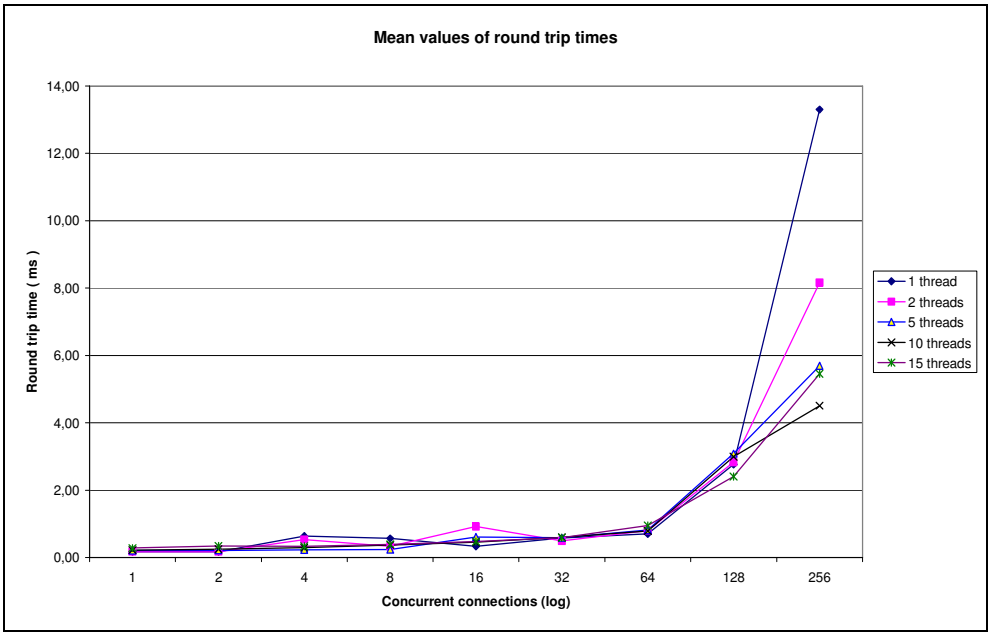

Figure 11: Mean values of round trip times without delay

Influence of amount of used threads can be seen much better when 1ms simulated processing delay is added to the system (Figure 12). Benefits of using multithreading with event driven architecture can be seen evidently. Already 5 threads in the thread pool decreases average round trip time significantly when amount of concurrent connections has increased over 32 connections.

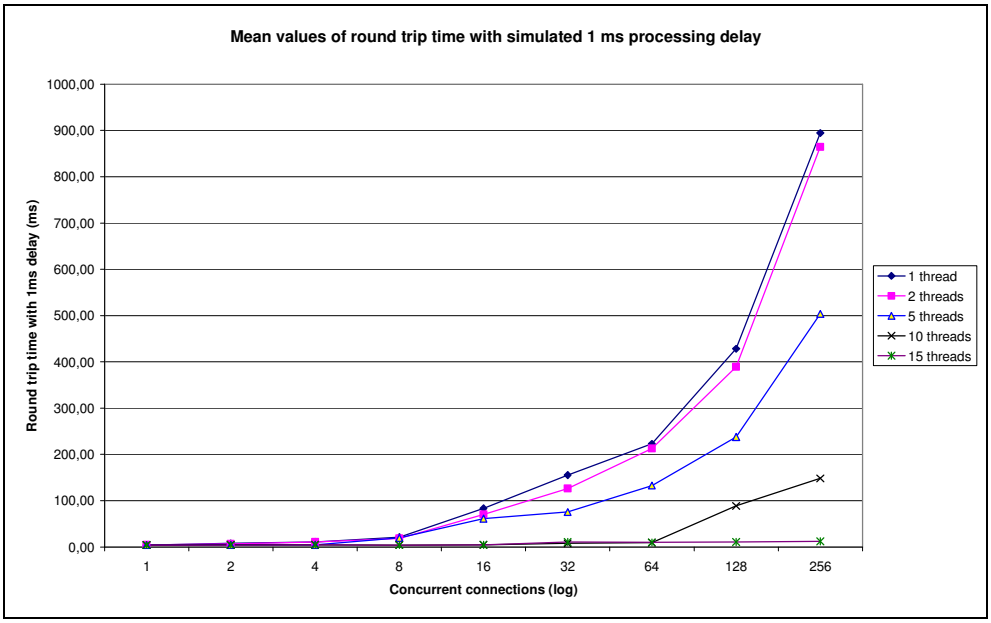

Figure 12: Round trip times with 1 ms processing delay

4.1.4 Results of the PeerHood Implementations Benchmarks

The PeerHood implementations were measured nearly in the way as the reference implementation. Test service for the earlier implementation was made so that it uses thread per connection model. When the PeerHood library transfer established connection to the test application new thread is created for that.

Results of the PeerHood benchmarking were very similar than reference implementation (Figure 13). Round trip times in preceding implementation stay unchanged although concurrent connections increase. In addition, thread per connection uses as many threads as concurrent connections are connected – which is not efficient.

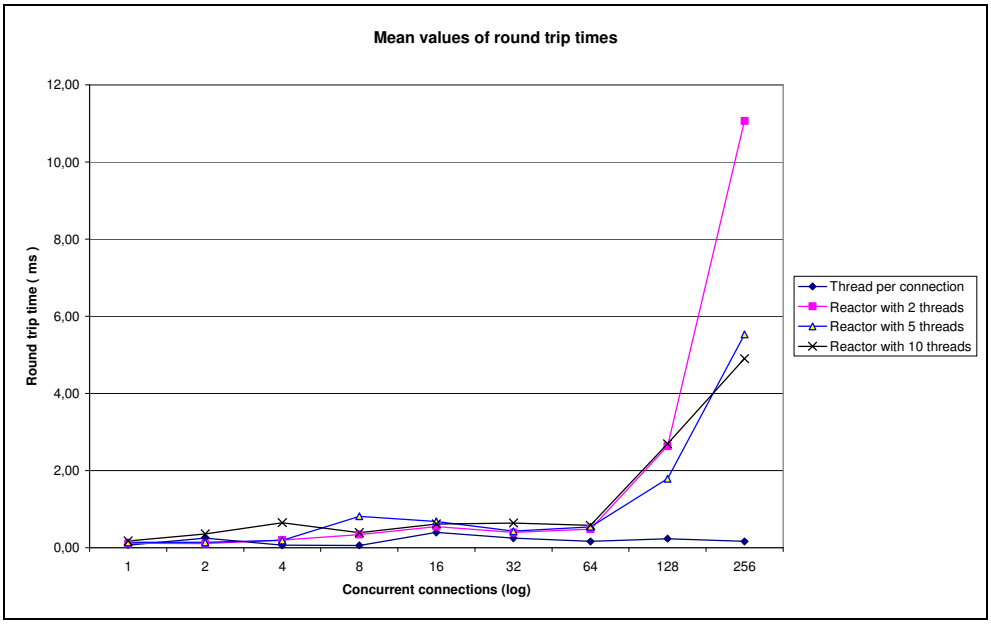

Figure 13: PeerHood round trip times

From Figure 14 are shown results of the benchmark with 1ms delay. From it can be noticed that in small amounts of concurrent connections differences between the Reactor and the thread per connection model are low. For higher number of concurrent connections thread pool solution with suitable amount of threads in the thread pool is still feasible.

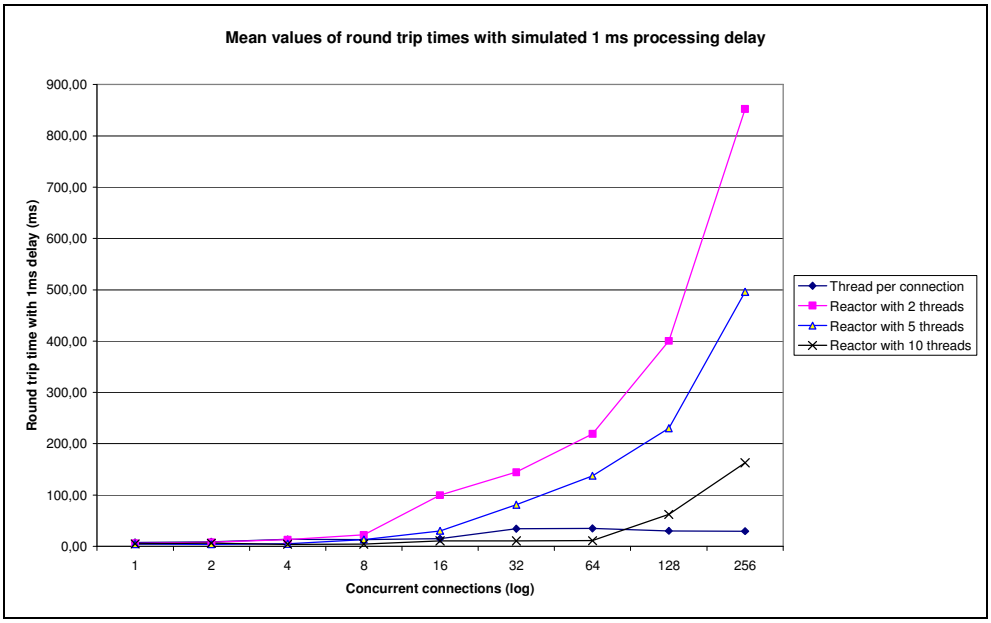

Figure 14: PeerHood round trip times with 1ms processing delay

From benchmarking results can be seen how important good analysis of needed thread amount in the thread pool is. Seeing that too small amount of threads will cause poor system performance and too many threads will cause unnecessary resource usage. Solutions can be traffic analysis – which is unfeasible in case of middleware where traffic depends of the application. Dynamic allocation for the threads in the thread pool can increase resource usage [14].

5. CONCLUSIONS

In low capacity devices like mobile phones resources are very limited. The thread per connection model uses much of resources and cause additional overhead by context switch. Reason for this study was implement and evaluate usability of the Reactor with thread pool approach to the PeerHood middleware. The Leader/Followers design pattern was selected as the thread pool model with the Reactor and with this combination performance evaluation was finally done.

Results from the benchmarks were very promising and in light of the results the Reactor with the Leader/Followers pattern seems to be useful model. With controlling amounts of threads in the thread pool used resources can be controlled easily. Challenge is to find right thread amount where used resources and system performance is in the good balance. For further study dynamic thread pool model might be worthwhile.

The Reactor design pattern is as well very dynamic model where new functionality is easy to add by adding new event handlers without causing changes to the existing model. Only small changes are needed to be done in the existing system if the Reactor is extended to support other event types.

REFERENCES

1. Douglas Schmidt, Michael Stal, Hans Rohnert and Frank Buschmann. Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects. Volume 2. John Wiley & Sons. 2000. 633 pages. ISBN 0471606952
2. Matt Welsh, Steven D. Gribbli, Eric A Brewer, David Culler. A Design Framework for Highly Concurrent Systems [web document]. University of California at Berkeley. 2000 [referred 22.4.2009]. Available from http://www.eecs.berkeley.edu/Pubs/TechRpts/2000/CSD-00-1108.pdf
3. Jari Porras, Petri Hiirsalmi, Ari Valtaoja. Peer-to-peer Communication Approach for a Mobile Environment. 37th IEEE Annual Hawaii International Conference on System Sciences. 2004. ISBN- 0-7695-2056-1
4. Douglas Schmidt, Chris Cleeland. Applying patterns to develop extensible ORB middleware. Communications Magazine. 1999. Vol. 37: 4. 54 - 63 pages. ISSN 0163-6804
5. Richard W. Stevens, Stephen A Rago, Advanced Programming in the Unix environment. Second edition. Addison-Wesley. 2005. 927 pages. ISBN 0-201-43307-9
6. Erich Gamma, Richard Helm, Ralph Jonson, John Vlissides, Design patterns – Elements of Reusable Object-Oriented software. Addison-Wesley. 1995. 416 pages. ISBN 0-201-63361-2
7. Swapna Gokhale et.al. Performance Analysis of the Reactor Pattern in Network Services. Parallel and Distributed Processing Symposium. 2006. 1 – 9 pages
8. Douglas C. Schmidt. Evaluating architectures for multithreaded object request brokers. Communications of the ACM archive. 1998. Vol. 41: 10. 54 – 60 pages. ISSN 0001-0782
9. Yibei Ling, Tracy Mullen, Xiaola Lin. Analysis of optimal thread pool size. Association for Computing Machinery. 2000. Vol. 34: 2. 42-55 pages. ISSN 0163-5980
10. Irfan Pyarali, Marina Spivak, Ron Cytron, Douglas C. Schmidt. Evaluating and Optimizing Thread Pool Strategies for Real-Time CORBA. Proceedings of the 2001 ACM SIGPLAN workshop on Optimization of middleware and distributed systems. 2001. 214 – 222 pages. ISBN 1-58113-426-6

11. Douglas C. Schmidt, Carlos O'Ryan, Irfan Pyarali, Michael Kircher, Frank Buschmann. Leader/Followers: A Design Pattern for Efficient Multi-threaded Event Demultiplexing and Dispatching. In Proceedings of the 6th Pattern Languages of Programming Conference. 2000. 1 – 29 pages
12. Irfan Pyarali. Patterns for providing real-time guarantees in doc middleware. Washington University. 2002. 100 pages. ISBN 0-493-84282-9
13. Maemo Software Platform. http://maemo.org/intro/platform/ [referred 23.3.2009]
14. DongHyun Kang, Saeyoung Han, SeoHee Yoo, Sungyong Park. Prediction-Based Dynamic Thread Pool Scheme for Efficient Resource Usage. IEEE 8th International Conference on Computer and Information Technology Workshops. 2008. 159 – 164 pages
15. Meamo SDK releases. http://maemo.org/development/sdks/maemo_4-1-2_diablo/ [referred 18.4.2009]
16. Scratchbox scross-compiling toolkit. http://www.scratchbox.org/ [referred 18.4.2009]

APPENDIX 1: REACTOR WITH LEADER/FOLLOWERS THREAD POOL

Reactor.h

```
/**
 *
 *  @project : Reactor design pattern example
 *  @File Name : Reactor.h
 *
 */
#ifndef REACTOR_H_
#define REACTOR_H_

// INCLUDES
#include <vector>
#include <sys/select.h>
#include <pthread.h>
#include "EventType.h"

// FORWARD DECLARATIONS
class EventHandlerArrayItem;
class EventHandler;
class Handle;
class ThreadPool;

/**
 * Reactor
 * Reactor class provides an interface for registering and removing event
 * handlers. Reactor class notifies event handlers for their requested
 * events
 */
class Reactor
{
public: // Reactor pattern interface
        /**
         * RegisterHandler
         * Register handler for register given EventHandler to receive given
         * eventtype. If Register Handler wants receive multiple events it
         * call several times this method using different EventTypes
         *
         * @param EventHandler& aEventHandler, reference for observing event
         *         handler
         * @param EventType aEventType, indicates which EventTypes
         *        EventHandler is interested
         */
    void RegisterHandler( EventHandler& aEventHandler,
                          EventType aEventType );

    /**
     * RemoveHandler
     * Method removes given EventType listed to given EventHandler
     * if there is not any EventTypes registered after remove then whole
     * EventHandler is removed
     *
     * @param EventHandler& aEventHandler
     * @param EventType aEventType
     */
    void RemoveHandler( EventHandler& aEventHandler,
                        EventType aEventType );

    /**
     * RemoveHandler
     * Removes EventHandler from list to receive events
     * @param EventHandler& aEventHandler
     */
    void RemoveHandler( EventHandler& aEventHandler );

    /**
     * HandleEvents
     * Method start watching events from event demultiplexing function
     */
    void HandleEvents( );
```

```
        /**
         * SetThreadPool
         * Method for setting thread pool
         * Thread pool reference is needed to promote new leader when needed
         * Ownership is transfered to reactor
         * @param ThreadPool* aPool
         */
        void SetThreadPool( ThreadPool* aPool );

public: // Singleton pattern
        /**
         * Instance
         * method for getting singleton object for Reactor class
         * In normal case it is enough one reactor in the process
         * @return Reactor* singleton instance for Reactor class
         */
        static Reactor* Instance();

        /**
         * c++ default destructor.
         * design pattern implementation
         */
        virtual ~Reactor();

protected:
        /**
         * ConvertToFdSets
         * Add registered handles to correct handle sets
         * @return highest descriptor number
         */
        virtual int ConvertToFdSets(  fd_set* aReadSet,
                                      fd_set* aWriteSet,
                                      fd_set* aExceptionSet );

        /**
         * FindHandler
         * @param EventHandler& aEventHandler
         * @return int index of element if it is found, else  < 0
         */
        virtual int FindHandler( EventHandler& aEventHandler );

        /**
         * FindHandler
         * @param Handle& aHandle
         * @return int index of element if it is found, else < 0
         */
        virtual int FindHandler( Handle& aHandle );

        /**
         * Notify
         * method for notifying select method to return. This is needed
         * if reactor's data has change during waiting event
         */
        virtual void Notify();

        /**
         * IsNotifier
         * Helper method to check is event source notifier
         * @param fd_set* aReadSet read handle, where notifier handle is set
         * @return bool, true if event source is from notifier
         */
        virtual bool IsNotifier( fd_set* aReadSet );

        /**
         * ResolveEventHandler
         * @param EventType& aOccuredEvent, parameter is for returning
         *        occured eventtype
         * @param fd_set* aReadSet, read handle set
         * @param fd_set* aWriteSet, write handle set
         * @param fd_set* aExceptionSet, exception handle set
         * @return EventHandler*, returns pointer to event handler which is
         *                response for handling occured event
         */
         virtual EventHandler* ResolveEventHandler(  EventType& aOccuredEvent,
                                                     fd_set* aReadSet,
                                                     fd_set* aWriteSet,
                                                     fd_set* aExceptionSet );
```

```
        /**
         * DeActivateHandle
         * Method for deactivate event type from given handle.
         * Event type needs to deactivate when it is occured.
         * By deactivating race condition between thread to race same
         * handle can be avoided
         * @param Handle aHandle, occured handle where event need to
         *        deactivate
         * @param EventType aEvent, is type which need to deactivate
         */
        virtual void DeActivateHandle( Handle aHandle, EventType aEvent );

        /**
         * ActivateHandle
         * When occured event is processed it can be activated again
         * @param Handle aHandle, occured handle where event need to
         *               activate
         * @param EventType aEvent, is type which need to activate
         */
        virtual void ActivateHandle( Handle aHandle, EventType aEvent );

        /**
         * CheckEvent
         * Helper method for checking is event occured
         * @param EventHandlerArrayItem& aEntity
         * @param fd_set* aSet
         * @param EventType aEvent
         */
        virtual bool CheckEvent( EventHandlerArrayItem& aEntity,
                                 fd_set* aSet,
                                 EventType aEvent );

private:
        /**
         * c++ default constructor. Method is private because of Singleton
         * design pattern implementation
         */
        Reactor();

private: // data

    /**
     * Array for storing registered handles
     * Array items are owned
     */
    std::vector<EventHandlerArrayItem*> *iItemArray;

    /**
     * Reactor data syncronizer for providing reentrancy
     */
    pthread_mutex_t* iDataSyncronizer;

    /**
     * descriptor for writing to pipe to notify select method if
     * internal data has changed
     */
    int iNotifierFd;

    /**
     * descriptor for reading pipe which is for notifying select method
     * to return
     */
    int iNotifiedFd;

    /**
     * reference to thread pool
     */
    ThreadPool* iThreadPool;

    /**
     * static Reactor* iInstance
     * attribute for singleton implementation
     */
    static Reactor* iInstance;
};
#endif  // REACTOR_H_
```

Reactor.cpp

```
/**
 *
 *  @project : Reactor design pattern example
 *  @File Name : Reactor.cpp
 *
 */

// INCLUDES
#include "Reactor.h"

#include <unistd.h>

#include "EventHandler.h"
#include "EventHandlerArrayItem.h"
#include "ThreadPool.h"
#include "MutexGuard.h"

//=============================================================================
// Reactor::RegisterHandler
//=============================================================================
void Reactor::RegisterHandler(EventHandler& aEventHandler,
                              EventType aEventType )
        {
        // protect internal data, mutex is locked from constructor
        // and unlocked from destructor
        MutexGuard guard( iDataSyncronizer );

        EventHandlerArrayItem* temp = NULL;
        int index = -1;

        index = FindHandler( aEventHandler );
        if( index < 0 )
                {
                temp = new EventHandlerArrayItem( aEventHandler, aEventType );
                iItemArray->push_back( temp );
                }
        else
                {
                temp = iItemArray->at( index );
                temp->SetEventType( aEventType );
                }
        Notify();
        }

//=============================================================================
// Reactor::RemoveHandler
//=============================================================================
void Reactor::RemoveHandler(  EventHandler& aEventHandler,
                              EventType aEventType )
        {
        // protect internal data, mutex is locked from constructor
        // and unlocked from destructor
        MutexGuard guard( iDataSyncronizer );

        EventHandlerArrayItem* temp = NULL;
        int index = -1;

        index = FindHandler( aEventHandler );
        if( index >= 0 )
                {
                temp = iItemArray->at( index );
                // if RemoveEventType return true, any event isn't left for
                // listening so it can remove
                if( temp->RemoveEventType( aEventType ) )
                        {
                        iItemArray->erase( iItemArray->begin() + index );
                        }
                }
        Notify();
        }

//=============================================================================
// Reactor::RemoveHandler
//=============================================================================
```

```
void Reactor::RemoveHandler( EventHandler& aEventHandler )
        {
        // protect internal data, mutex is locked from constructor
        // and unlocked from destructor
        MutexGuard guard( iDataSyncronizer );

        int index = -1;
        index = FindHandler( aEventHandler );
        if( index >= 0 )
                {
                iItemArray->erase(iItemArray->begin() + index );
                }
        Notify();
        }

//============================================================================
// Reactor::HandleEvents
// Method is heart of reactor pattern. It uses select method to
// demultiplexing events. When event occurs first search correct
// EventHandler and then notify that for event
//============================================================================
void Reactor::HandleEvents( )
        {
        int numberOfEvents = 0;
        int highestHandleNumber = 0;

        // handles
        fd_set readSet;
        fd_set writeSet;
        fd_set exceptionSet;

        // handle sets need to be zeroed before those are setted
        FD_ZERO( &readSet );
        FD_ZERO( &writeSet );
        FD_ZERO( &exceptionSet );

        // add all event handlers to those handle set,
        // where they are registered
        highestHandleNumber = ConvertToFdSets(&readSet,
                                              &writeSet,
                                              &exceptionSet );


        numberOfEvents = select( highestHandleNumber + 1,
                                 &readSet,
                                 &writeSet,
                                 &exceptionSet,
                                  0 );

        // if notifier, no need to go through all eventhandlers anymore
        // IsNotifier promotes new leader
        if( IsNotifier( &readSet ) )
                {
                return;
                }

        // if error occurs
        if( numberOfEvents <= 0 )
                {
                // Handle timeout
                // Handle errors
                if( iThreadPool )
                        {
                        iThreadPool->PromoteNewLeader();
                        }
                return;
                }

        // find event handler and occured event type
        EventType occuredEvent;
        EventHandler* eventHandler = ResolveEventHandler( occuredEvent,
                                                          &readSet,
                                                          &writeSet,
                                                          &exceptionSet );
        if( eventHandler )
                {
                // Deactivate event from Handle
```

```
                // Can't trust that eventhandler is exsist when HandleEvent
                // returns
                Handle handle = *(eventHandler->GetHandle());

                 DeActivateHandle( handle, occuredEvent );

                if( iThreadPool )
                        {
                        iThreadPool->PromoteNewLeader();
                        }

                eventHandler->HandleEvent( occuredEvent );

                ActivateHandle( handle, occuredEvent );

                }
        else // unkonw event happeded, promote new leader and return back
                {
                if( iThreadPool )
                        {
                        iThreadPool->PromoteNewLeader();
                        }
                }
        }

//=============================================================================
// Reactor::SetThreadPool
//=============================================================================
void Reactor::SetThreadPool( ThreadPool* aPool )
        {
        iThreadPool = aPool;
        }

//=============================================================================
// Reactor::Instance
//=============================================================================
Reactor* Reactor::iInstance = NULL;
Reactor* Reactor::Instance()
        {
        if( !iInstance )
                {
                iInstance = new Reactor();
                }
        return iInstance;
        }

//=============================================================================
// Reactor::~Reactor
//=============================================================================
Reactor::~Reactor()
    {
   if( iThreadPool )
        {
        iThreadPool->Stop();
        }

    if( iItemArray )
        {
        iItemArray->clear();
        delete iItemArray;
        }

    if( iDataSyncronizer )
        {
        pthread_mutex_destroy( iDataSyncronizer );
        delete iDataSyncronizer;
        }

    if( iNotifierFd > 0 )
        {
        close( iNotifierFd );
        }

    if( iNotifiedFd > 0 )
        {
        close( iNotifiedFd );
        }
```

```
    if( iThreadPool )
        {
        delete iThreadPool;
        iThreadPool = NULL;
        }
    }

//=============================================================================
// Reactor::Reactor
//=============================================================================
Reactor::Reactor()
: iItemArray( NULL ),
  iDataSyncronizer( NULL ),
  iNotifierFd( 0 ),
  iNotifiedFd( 0 ),
  iThreadPool( NULL )
        {
        iItemArray = new std::vector<EventHandlerArrayItem*>();
        iDataSyncronizer = new pthread_mutex_t;

        if( iDataSyncronizer )
                {
                pthread_mutex_init( iDataSyncronizer, NULL );
                }

        // For notifying select function to return if reactor
        // event handler data has changed
        int fd[2];
        if( pipe( fd ) == 0 )
                {
                iNotifiedFd = fd[0]; // read pipe
                iNotifierFd = fd[1]; // write pipe
                }
        }

//=============================================================================
// Reactor::ConvertToFdSets
//=============================================================================
int Reactor::ConvertToFdSets( fd_set* aReadSet,
                              fd_set* aWriteSet,
                              fd_set* aExceptionSet )
        {
        // protect internal data, mutex is locked from constructor
        // and unlocked from destructor
        MutexGuard guard( iDataSyncronizer );

        int highestDescriptor = -1;

        EventHandlerArrayItem* temp = NULL;
        int fd_temp = 0;

        for ( unsigned int i = 0; i < iItemArray->size(); i++ )
                {
                temp = iItemArray->at( i );
                fd_temp = temp->Handler()->GetHandle()->GetHandle();

                if( fd_temp > highestDescriptor )
                        {
                        highestDescriptor = fd_temp;
                        }

                /*
                 * If read event is set or accept event, add it to read
               * descriptor set. In select point of view these are same thing
                 */
                if(temp->IsEvent( EReadEvent )||temp->IsEvent( EAcceptEvent ))
                        {
                        FD_SET( fd_temp , aReadSet );
                        }

                /*
                 * if write event is set, add it to write descriptor set
                 */
                if( temp->IsEvent( EWriteEvent ) )
                        {
                        FD_SET( fd_temp, aWriteSet );
```

```
                        }

                /*
                 * if exception event is set, add it to exception descriptor
               * set
                 */
                if( temp->IsEvent( EExceptionEvent ) )
                        {
                        FD_SET( fd_temp, aExceptionSet );
                        }
                }

                // finally add notify descriptor
                if( iNotifiedFd > 0 )
                        {
                        FD_SET( iNotifiedFd, aReadSet );
                        if( iNotifiedFd > highestDescriptor )
                                {
                                highestDescriptor = iNotifiedFd;
                                }
                        }
        return highestDescriptor;
        }

//============================================================================
// Reactor::FindHandler
//============================================================================
int Reactor::FindHandler( EventHandler& aHandler )
        {
        return FindHandler( *(aHandler.GetHandle()) );
        }

//============================================================================
// Reactor::FindHandler
//============================================================================
int Reactor::FindHandler( Handle& aHandle )
        {
        for( unsigned int i = 0; i < iItemArray->size(); i++ )
                {
                if( iItemArray->at( i )->Compare( aHandle ) )
                        {
                        return i;
                        }
                }
        return -1;
        }

//============================================================================
// Reactor::Notify
//============================================================================
void Reactor::Notify()
        {
        if( iNotifierFd > 0 )
                {
                write( iNotifierFd, "1", sizeof(char));
                }
        }

//============================================================================
// Reactor::IsNotifier
//============================================================================
bool Reactor::IsNotifier( fd_set* aReadSet )
        {
        MutexGuard guard( iDataSyncronizer );

        if( FD_ISSET( iNotifiedFd, aReadSet) )
                {
                char buf;
                read( iNotifiedFd, &buf, sizeof(char) );
                iThreadPool->PromoteNewLeader();
                return true;
                }
        else
                {
                return false;
                }
        }
```

```
//============================================================================
// Reactor::ResolveEventHandler
//============================================================================
EventHandler* Reactor::ResolveEventHandler(  EventType& aOccuredEvent,
                                             fd_set* aReadSet,
                                             fd_set* aWriteSet,
                                             fd_set* aExceptionSet )
        {
        MutexGuard guard( iDataSyncronizer );

        EventHandlerArrayItem* temp = NULL;

        //now resolve handles which cause event
        for( unsigned int i = 0; i < iItemArray->size(); i++ )
                {
                temp = iItemArray->at( i );
                if( CheckEvent( *temp, aReadSet, EReadEvent ) )
                        {
                        aOccuredEvent = EReadEvent;
                        return temp->Handler();
                        }
                else if( CheckEvent( *temp, aWriteSet, EWriteEvent ) )
                        {
                        aOccuredEvent = EWriteEvent;
                        return temp->Handler();
                        }
                else if( CheckEvent( *temp, aExceptionSet, EExceptionEvent ) )
                        {
                        aOccuredEvent = EExceptionEvent;
                        return temp->Handler();
                        }
                }
        //Unknow reason
        return NULL;
        }

//============================================================================
// Reactor::DeActivateHandle
//============================================================================
void Reactor::DeActivateHandle( Handle aHandler, EventType aEvent )
        {
        MutexGuard guard( iDataSyncronizer );

        int index = FindHandler( aHandler );
        if( index >= 0 )
                {
                iItemArray->at( index )->DeActivateEvent( aEvent );
                // no need to notify, Deactivation is done before leader
                // promotion
                }
        }

//============================================================================
// Reactor::ActivateHandle
//============================================================================
void Reactor::ActivateHandle( Handle aHandler, EventType aEvent )
        {
        MutexGuard guard( iDataSyncronizer );
        int index = FindHandler( aHandler );
        if( index >= 0 )
                {
                iItemArray->at( index )->ActivateEvent( aEvent );
                Notify();
                }
        }

//============================================================================
// Reactor::CheckEvent
//============================================================================
bool Reactor::CheckEvent( EventHandlerArrayItem& aEntity,
                          fd_set* aSet,
                          EventType aEvent )
        {
        bool retVal = false;
        /*
         * fd_set need to check before, event check. This way
```

```
         * all event will be handled in somehow
         */
        if( FD_ISSET( aEntity.Handler()->GetHandle()->GetHandle(), aSet ) )
                {
                // ensure that event handler wait occurred event
                if( aEntity.IsEvent( aEvent ) )
                        {
                        retVal = true;
                        }
                }
        return retVal;
        }

// End of file
```

EventHandler.h

```
/**
 *
 *  @project : Reactor design pattern example
 *  @File Name : EventHandler.h
 *
 */
#ifndef EVENTHANDLER_H_
#define EVENTHANDLER_H_

// INCLUDES
#include "EventType.h"
#include "Handle.h"

/**
 * EventHandler
 * is abstraction class for receiving Events send by Reactor class and
 * class provides access to it's handle which is used to event binding
 */
class EventHandler
{
public:
        /**
         * c++ default destructor
         */
        virtual ~EventHandler(){};

        /**
         * HandleEvent
         * pure virtual interface method for receiving observed events from
         * Reactor
         * @param EventType aEventType, occurred event type
         */
        virtual void HandleEvent( EventType aEventType ) = 0;

        /**
        * GetHandle
        * @return Handle*, pointer for Handle which events we want to
        * receive. ownership is not transfered
        */
        virtual Handle* GetHandle();

protected:

        /**
         * Handle* iHandle
         * pointer for Handle
         * owned
         */
        Handle* iHandle;
};
#endif  // EVENTHANDLER_H_
```

Handle.h

```
/**
 *
 *  @project : Reactor design pattern example
 *  @File Name : Handle.h
 *
 */
#ifndef HANDLE_H_
#define HANDLE_H_

/**
 * Handle
 * class is wrapper for real OS file/socket/pipe handle
 */
class Handle
{
public:
        /**
         * c++ default constructor
         * @param int aHandle, OS handle
         */
        Handle( int aHandle );

        /**
         * c++ default destructor
         */
        virtual ~Handle();

        /**
         * GetHandle
         * @return int OS handle
         */
        int GetHandle() const;

        /**
         * operator ==
         */
        bool operator==( const Handle& aCompare );

private:
        /**
         * int iHandle
         * Wrapped handle
         */
        int iHandle;
};
#endif  // HANDLE_H_
```

Handle.cpp

```
/**
 *
 *  @project : Reactor design pattern example
 *  @File Name : Handle.cpp
 *
 */

#include "Handle.h"

//=============================================================================
// Handle::Handle
//=============================================================================
Handle::Handle( int aHandle )
: iHandle( aHandle )
        {
        }

//=============================================================================
// Handle::~Handle
//=============================================================================
Handle::~Handle()
```

```
        {
        }

//=========================================================================
// Handle::GetHandle
//=========================================================================
int Handle::GetHandle() const
        {
        return iHandle;
        }

//=========================================================================
// Handle::operator==
//=========================================================================
bool Handle::operator==( const Handle& aCompare )
        {
        return iHandle == aCompare.iHandle;
        }

// End of file
```

EventHandlerArrayItem.h

```
/**
 *
 *  @project : Reactor design pattern example
 *  @File Name : EventHandlerArrayItem.h
 *
 */
#ifndef EVENTHANDLERARRAYITEM_H_
#define EVENTHANDLERARRAYITEM_H_

//INCLUDES
#include "EventType.h"

// FORWARD DECLARATIONS
class EventHandler;
class Handle;

/**
 * EventHandlerArrayItem
 * Array item structure for grouping together EventHandler and associated
 * EventTypes for that
 */
class EventHandlerArrayItem
{
public:
        /**
         * EventHandlerArrayItem
         */
        EventHandlerArrayItem(EventHandler& aEventHandler, EventType aEvent);

        /**
         * ~EventHandlerArrayItem
         */
        virtual ~EventHandlerArrayItem();

        /**
         * Handler
         */
        EventHandler* Handler();

        /**
         * IsEvent
         */
        bool IsEvent( EventType aEventType );

        /*
         * Compare
         */
        bool Compare( EventHandler& aCompare );
```

```
        /*
         * Compare
         */
        bool Compare( Handle& aCompare );

        /**
         * SetEventType
         * @param EventType aEventType
         */
        void SetEventType( EventType aEventType );

        /**
         * DeActivateEvent
         */
        void DeActivateEvent( EventType aEventType );

        /**
         * ActivateEvent
         */
        void ActivateEvent( EventType aEventType );

        /**
         * RemoveEventType
         * @param EventType aEventType
         * @return bool, inidcates is any event left or not
         */
        bool RemoveEventType( EventType aEventType );

protected:

        /**
         * EventHandlerArrayItem
         */
        EventHandlerArrayItem();


        /**
         * Registered event handler
         */
        EventHandler* iHandler;

        /**
         * Registered event types
         */
        int iEventType;

        /**
         * Event which are deactivated
         */
        int iDeactivatedEvent;
};
#endif /* _EVENTHANDLERARRAYITEM_H_ */
```

EventHandlerArrayItem.cpp

```
/**
 *
 *  @project : Reactor design pattern example
 *  @File Name : EventHandlerArrayItem.cpp
 *
 */

#include "EventHandlerArrayItem.h"
#include "EventHandler.h"
#include "EventType.h"

//=============================================================================
// EventHandlerArrayItem::EventHandlerArrayItem
//=============================================================================
EventHandlerArrayItem::EventHandlerArrayItem()
: iHandler( 0 ), iEventType( 0 ), iDeactivatedEvent( 0 )
        {
        }
```

```
//============================================================================
// EventHandlerArrayItem::EventHandlerArrayItem
//============================================================================
EventHandlerArrayItem::EventHandlerArrayItem( EventHandler& aEventHandler,
                                              EventType aEvent )
: iHandler( &aEventHandler ), iEventType( aEvent ), iDeactivatedEvent( 0 )
        {
        }

//============================================================================
// EventHandlerArrayItem::~EventHandlerArrayItem
//============================================================================
EventHandlerArrayItem::~EventHandlerArrayItem()
        {
        //iHandler not owned
        }

//============================================================================
// EventHandlerArrayItem::Handler
//============================================================================
EventHandler* EventHandlerArrayItem::Handler()
        {
        return iHandler;
        }

//============================================================================
// EventHandlerArrayItem::IsEvent
//============================================================================
bool EventHandlerArrayItem::IsEvent( EventType aEventType )
        {
        bool ret = false;
        if( iEventType & aEventType )
                {
                ret = true;
                }
        if( iDeactivatedEvent & aEventType )
                {
                ret = false;
                }

        return ret;
        }

//============================================================================
// EventHandlerArrayItem::Compare
//============================================================================
bool EventHandlerArrayItem::Compare( EventHandler& aCompare )
        {
        return Compare( *(aCompare.GetHandle()) );
        }

//============================================================================
// EventHandlerArrayItem::Compare
//============================================================================
bool EventHandlerArrayItem::Compare( Handle& aCompare )
        {
        return *( iHandler->GetHandle() ) == aCompare;
        }

//============================================================================
// EventHandlerArrayItem::SetEventType
//============================================================================
void EventHandlerArrayItem::SetEventType( EventType aEventType )
        {
        iEventType |= aEventType;
        }

//============================================================================
// EventHandlerArrayItem::ActivateEvent
//============================================================================
void EventHandlerArrayItem::ActivateEvent( EventType aEventType )
        {
        iDeactivatedEvent &= ~aEventType;
        }
```

```
//=============================================================================
// EventHandlerArrayItem::DeActivateEvent
//=============================================================================
void EventHandlerArrayItem::DeActivateEvent( EventType aEventType )
        {
        iDeactivatedEvent |= aEventType;
        }

//=============================================================================
// EventHandlerArrayItem::RemoveEventType
//=============================================================================
bool EventHandlerArrayItem::RemoveEventType( EventType aEventType )
        {
        iEventType &= ~aEventType;
        if( iEventType == 0 )
                {
                return true;
                }
        return false;
        }

// End of file
```

ThreadPool.h

```
/**
 *
 *  @project : Reactor and Leader/Followers design pattern example
 *  @File Name : ThreadPool.h
 *
 */
#ifndef THREADPOOL_H_
#define THREADPOOL_H_

// INCLUDES
#include <vector>
#include <pthread.h>

// FORWARD DECLARATIONS
class Reactor;

/**
 * Class implement main part of Leader / Followers design pattern
 */
class ThreadPool
{
public:
        /**
         * class constructor
         * @param Reactor& reference to Reactor object
         * @param unsigned int aThreadCount
         */
        ThreadPool( Reactor& aReactor, unsigned int aThreadCount );

        /**
         * c++ default destructor
         */
        virtual ~ThreadPool();

        /**
         * Join
         * Thread use this to join the pool
         */
        void Join();

        /**
         * PromoteNewLeader
         * Method for promote new leader
         */
        void PromoteNewLeader();
```

```
        /**
         * Method will start demultiplexing,
         * first it create needed threads and starts handling events
         */
        void Start();

        /**
         * Method will stop demultiplexing
         */
        void Stop();

private:

        /**
         * Helper method to provide callback from thread creation
         */
        static void* WorkerThread( void* aArg );

private:

        // Reactor reference
        Reactor& iReactor;

        // How many thread are in the pool
        unsigned int iNumOfThreads;

        // id list of all thread in the pool
        std::vector<pthread_t> iThreadidps;

        // identify which thread is leader thread
        pthread_t iLeaderThread;

        // internal data syncronizer
        pthread_mutex_t* iSyncronizer;

        // syncronizer for leader followers
        pthread_cond_t* iLFCondSyncronizer;

        bool iIsRunning;
};
#endif // THREADPOOL_H_
```

ThreadPool.cpp

```
/**
 *
 *  @project : Reactor and Leader / Followers design pattern example
 *  @File Name : ThreadPool.cpp
 *
 */

#include "ThreadPool.h"
#include "MutexGuard.h"
#include "Reactor.h"

// CONSTANTS
const unsigned int KNoLeader = 0;

//=============================================================================
// ThreadPool::ThreadPool
//=============================================================================
ThreadPool::ThreadPool( Reactor& aReactor, unsigned int aThreadCount )
:iReactor( aReactor ),
 iNumOfThreads( aThreadCount ),
 iThreadidps( aThreadCount ),
 iLeaderThread( KNoLeader ),
 iIsRunning( false )
    {
    iSyncronizer = new pthread_mutex_t;
    pthread_mutex_init( iSyncronizer, NULL );

    iLFCondSyncronizer = new pthread_cond_t;
    pthread_cond_init( iLFCondSyncronizer, NULL );
```

```
    iReactor.SetThreadPool( this );
    }

//=============================================================================
// ThreadPool::~ThreadPool
//=============================================================================
ThreadPool::~ThreadPool()
    {
    iIsRunning = false;

    pthread_cond_broadcast( iLFCondSyncronizer );
    pthread_cond_destroy( iLFCondSyncronizer );
    delete iLFCondSyncronizer;

    pthread_mutex_destroy( iSyncronizer );
    delete iSyncronizer;

    // cancel created threads
    for( unsigned int i = 0; i < iNumOfThreads; i++ )
        {
        pthread_join( iThreadidps[i], NULL );
        }
    }

//=============================================================================
// ThreadPool::Join
//=============================================================================
void ThreadPool::Join( )
        {
        // mutex lock is aquired in MutexGuard constructor
        MutexGuard guard( iSyncronizer );

      while( iIsRunning )
                {
                while( iLeaderThread != KNoLeader )
                        {
                        pthread_cond_wait( iLFCondSyncronizer, iSyncronizer );
                        if( !iIsRunning )
                            {
                            return;
                            }
                        }

                // new leader promoted
                iLeaderThread = pthread_self();

                // now we can let new thread join safely if there is some
                // wating the mutex
                guard.UnLock();

                // start demultiplexing,
                // wait event to occur
                iReactor.HandleEvents( );

                // get mutex lock again
                guard.Lock();
                }
        }

//=============================================================================
// ThreadPool::PromoteNewLeader
//=============================================================================
void ThreadPool::PromoteNewLeader()
        {
        // mutex lock is aquired in MutexGuard constructor
        MutexGuard guard( iSyncronizer );

        // only leader thread can call this
        if( iLeaderThread != pthread_self() )
                {
                return;
                }

        iLeaderThread = KNoLeader;

        // notify leader promotion
```

```
        pthread_cond_signal( iLFCondSyncronizer );

        // mutex is unlocked in MutexGuard destructor
        }

//=============================================================================
// ThreadPool::Start
//=============================================================================
void ThreadPool::Start()
        {
    iIsRunning = true;
        for( unsigned int i = 0; i < iNumOfThreads; i++ )
                {
                pthread_create( &iThreadidps[i],
                                NULL,
                                ThreadPool::WorkerThread,
                                this );
                }
        }

//=============================================================================
// ThreadPool::Stop
//=============================================================================
void ThreadPool::Stop()
    {
    iIsRunning = false;
    }

//=============================================================================
// ThreadPool::WorkerThread
//=============================================================================
void* ThreadPool::WorkerThread( void* aArg )
        {
        if( !aArg )
                {
                return (void*)0;
                }
        ThreadPool* object = reinterpret_cast<ThreadPool*> (aArg);
        object->Join();
        return (void*)0;
        }

// End of file
```

MutexGuard.h

```
/**
 *
 *  @project : Reactor and Leader / Followers design pattern example
 *  @File Name : MutexGuard.h
 *
 */
#ifndef MUTEXGUARD_H_
#define MUTEXGUARD_H_

#include <pthread.h>

/**
 * Scoped locking design pattern
 */
class MutexGuard
{
public:
        /**
        * default constructor
        * mutex lock will acquired in the consructor
        * @param pthread_mutex_t mutex which is used
        */
        MutexGuard( pthread_mutex_t* aMutex )
        : iOwner( false ), iMutex( aMutex )
                {
                if( pthread_mutex_lock( iMutex ) == 0 )
```

```
                        {
                        iOwner = true;
                        }
                }

        /**
         * default destructor
         * mutex lock will be released in the destructor
         */
        ~MutexGuard()
                {
                if( iOwner )
                        {
                         pthread_mutex_unlock( iMutex );
                        }
                }

        /**
         * Lock
         * Get lock for the mutex explicitly
         */
        void Lock()
                {
                if( pthread_mutex_lock( iMutex ) == 0 )
                    {
                    iOwner = true;
                    }
                }


       /**
          * UnLock
          * get unlock for the mutex explicitly
          */
        void UnLock()
                {
                pthread_mutex_unlock( iMutex );
                iOwner = false;
                }

// avoid copying and assign
private:
        MutexGuard( const MutexGuard& aGuard );
        void operator= ( const MutexGuard& aGuard );

private: // data

        bool iOwner;
        pthread_mutex_t* iMutex;
};

#endif // MUTEXGUARD_H_
```