Universal Communication Component on Symbian Series60 Platform

Róbert Kereskényi, Bertalan Forstner, Hassan Charaf

Department of Automation and Applied Informatics
Budapest University of Technology and Economics
Goldmann György tér 3, H-1111 Budapest, Hungary
Robert.Kereskenyi@aut.bme.hu, Bertalan.Forstner@aut.bme.hu,
hassan@aut.bme.hu

Abstract: In the recent years a new category of handheld devices appeared. The so-called smartphone is a combination of the regular cell phones and PDAs. It is a fully functional hand sized compute that supports phone calls. As these devices became popular, more and more application was developed to the different smartphone platforms. These softwares benefit from the different communication technologies in order to be able to use them with other mobile or desktop applications.
There are different mobile platforms, and Symbian is the most promising among them. The number of applications available by downloading or by purchasing for Symbian OS is increasing every day.
The communication API of this platform was built on the Active Object design pattern to handle the asynchronous communication operations. We can reach each type of communication protocol through this API, but they differ from each other, and therefore they can be hard to implement, mainly when we want to do it in one application.
Therefore we designed and implemented an universal communication component on Symbian Series60 platform, that can be used as a linked library. It supports all available communication type through a common interface, hides the differences of the protocols from the developers, and easily configurable for each communication type.

Keywords: Design patterns, network communication, mobile devices, mobile development, Symbian

1 Introduction

In the last few years we can observe a rapid and continuous evolution of mobile communication systems. While the devices become smaller and smarter, their role in our life has been totally reinterpreted. At the beginning the phones were used only for voice calls, nowadays, however, they are much like universal communication tools.

Different devices with different platforms exist from several vendors. Almost all of these platforms are open to installing and running applications developed by independent software vendors. Therefore, voice transmission has become a secondary function, and other information management software has turned into the essential part of the platform.

Since mobile software development does not have a long history, the development culture has only a substandard, and is still evolving. The developers often need to solve recurring problems, invent the solution again and again in each application, which is very time consuming. In addition, it is often hard to reuse existing algorithms, interfaces or implementations due to the heterogeneous hardware and software architecture, or the different operating systems. The developers can not focus on the real problem; they need to deal with the subsidiary tasks. That makes development ineffective and error prone. Without a standard structure, the source code is hard to read. It is especially true when talking about mobile communication systems.

Design patterns [1] are the basis for the widespread reuse of software architectures and design solutions. There are several patterns in desktop environment we can use for typical problems that arise in implementing communicating applications. They could be used as a starting point for software design in mobile environment. However, the limited resources of the mobile devices need to be taken into account.

2 Related Work

Recall that in the last few years a new generation of mobile devices appeared. These devices were a combination of the existing cell phone and the personal digital assistants (PDAs). The smartphone – a fully functional computer in a cell phone shape and size – means new opportunities and also problems for developers that need to be solved. Both of them derive from the special properties and design of the device: smart and small.

The usability of the device based on its operating system, which is becoming smarter day by day. It can be considered an advantage or disadvantage, but there are different mobile operating systems [2]. They can not be regarded as downscaled desktop operating systems (OS), nor as embedded systems with communication capabilities. They are new types of operating systems developed to satisfy the special needs of mobile devices. The most frequent platforms are Symbian OS, Windows Mobile, and Mobile Linux. Similarly to the case of desktop environments, the applications are not easily portable between different operating systems.

The devices are designed to communicate with other desktop or mobile computers or servers. Therefore they are capable of long and short distance communications on wired and wireless barriers. Since the devices are mobile, in most cases the wireless communication techniques should be used. The applicable technologies are, for example, 3G UMTS, GPRS, IrDA or Bluetooth. Although they are totally different transport techniques, at the application layer they all can be handled with sockets. After initializing the current communication technology, there is a socket on the server and client side that is needed to be connected. There are different discovery services depending on the technology currently used to select the remote device before establishing the connection. On each side of the communication, there is a socket to read and write.

Communication is an asynchronous operation: when the data is ready to send, it is written to the socket, and when the data arrives, it must be read from the socket [3]. Therefore, it fits only into an event driven application structure.

We only focus on the part of an application that is responsible for communication. In most cases there is no activity, only waiting for data to send or receive. When the data arrives, the main application should be notified such that it can process the read data, and reader process listens again to the socket for receiving data. The main application process must be prevented to run into a waiting block. In desktop environments, there is another thread that can be blocked, which notifies the main application via a callback function when the data is read from the socket and can be processed.

We searched for a similar solution in mobile environment, taking care of the fact that much less resource is available, and starting a new thread is "expensive". The solutions can be best described with the following design patterns.

2.1 The Reactor Design Pattern

The Reactor architectural design pattern enables event-driven applications to demultiplex and dispatch request delivered to an application from one or more clients. [4][5]

The communication software must respond to each event generated by multiple sources. These events are from multiple I/O that are the different resources managed by the operating system. The events can occur from multiple devices simultaneously. Therefore a single-threaded application must not block on reading from a specified I/O channel, because events from other channels could not be handled. Multi-threaded application can be a potential solution for the problem: a separate thread should handle each connection. Each thread can be blocked in a READ call, since it does not have any affect on other peers, only the one it is associated. When an event arrives, the thread unblocks, event is handled, and the thread reblocks again. It could be a solution, but using multi-threading for event handling in communication applications has many disadvantages:

- Threading may cause complex concurrency control.
- Threading may decrease the performance due to data movement, synchronization, context switching.
- Threading is not available on all operating systems.

The Reactor pattern manages a single-threaded event loop that performs event demultiplexing and event handler dispatching in response to events from multiple sources.

The Reactor pattern can be used in the following cases,

- Multiple events may occur simultaneously from multiple resources, and blocking or continuous polling on an individual resource listening to an event is inefficient.
- Each event handler processes its messages in a short period of time.
- Using multi-threading for event handling in communication is impossible because of the absence of multi-threading support of the operating system.
- Using multi-threading for event handling in communication is inappropriate due to the poor performance or very high complexity it causes.

The collaboration in the described pattern is performed as follows:

- An application registers its own concrete event handler with the reactor.
- After all event handles are registered, the application starts the event loop of the reactor, called handle_events(). It then calls the synchronous event demultiplexer to wait for indication events to occur.
- The synchronous event demultiplexer function returns to the reactor when one handle corresponding to event sources becomes ‘ready’.
- The reactor dispatches the corresponding hook method of the appropriate event handler.

The advantages of using the Reactor pattern are the following:

- It improves the modularity and reusability of event-driven application software by separating the application-independent mechanisms from application-specific part.
- It improves application portability by reusing its interface independently from the operating system calls that perform event demultiplexing.

However, it also has some disadvantages:

- Event handlers are not preempted during execution. Therefore it should not block on an I/O, and should cause short-duration operations.
- Applications using Reactor pattern can be hard to debug because of their flow control.

2.2 The Active Object Design Pattern

The Active Object design pattern decouples the method execution from method invocation to enhance concurrency and simplify synchronized access to objects that reside in their own threads of control [4][6][7].

The context of usage is that clients access objects running in separate threads of control.

Many program benefit from using concurrent objects to improve their quality of service, for example by allowing an application to handle multiple client requests simultaneously. Instead of using single-threaded passive objects, which executes its methods in the thread of control of the client that invoked the methods, a concurrent object resides in its own thread of control. If objects run concurrently, access to their method and data must be synchronized if these objects are shared and modified by multiple client thread. In the presence of this problem the following forces arise:

- Methods invoked on an object concurrently should not block the entire process so that to prevent degrading the quality of service of other methods.
- Synchronized access to shared object should be as simple as possible.
- Applications should be designed to transparently leverage the parallelism available on a hardware and software platform.

A solution can be that each object that requires concurrent execution decouples method invocation on the object from method execution. Method invocation should occur in the client's thread of control, whereas method execution should occur in a separate thread. This decoupling should be designed such that the client thread appears to invoke an ordinary method.

There is an object (proxy) that acts as a well defined interface for an active object, and another object (servant) exists that implements this interface. These objects run in separate threads, thus, method invocation and method execution can run concurrently.

At run-time the proxy transforms the client method invocations of the client into method requests stored in an activation list by the scheduler. The event loop of the scheduler runs continuously in the same thread as the servant, dequeueing method request from activation list and dispatching them on the servant.

The benefits of the Active Object design pattern are the followings:

- It enhances application concurrency and simplifies the synchronization complexity.
- Transparently leverages available parallelism.
- Method execution order can differ from method invocation order.

The Active Object pattern has a few disadvantages:

- Performance overhead.
- Complicated debugging.

3 Series60 Universal Communication Component

As an example on the basics of an active object, a communication component can be developed and reused in many of further applications that incorporate communication. Using an existing solution during software development makes our life easier, therefore such an universal communication component is definitely efficient. We realized the solution on Symbian Series60 platform [8], but an idea can be applicable also on other platforms.

On the Symbian platform there are two different communication subsystems: serial and socket based. Both uses abstract classes over the bottom layers, which can be configured with strings or other protocol identifiers. Thus, the communication subsystem employs the currently selected protocol. Its most important advantage is that all the communication protocols are accessible on a common layer with different configurations. The supported technologies are IrDA, Bluetooth, and TCP/IP.

With the native support of the framework realizing all technologies in our application is still a challenge. Therefore we thought that such a universal communication component handles all technologies, provides the same interface for all protocols. Furthermore, all initialization phase has a default setting (that can be modified manually) that can be used in further application development. We only have to specify which technology we want to use, then connect to the remote device, perform, the data transfer, and finally disconnect. It is much easier to reuse than implement it every time.

The component must be able to handle point-to-.point communication with each type of protocols mentioned above, transfer data confidentially (error checking and repairing, keeping packet order), disconnect and release the resources. Furthermore, it must be easy-to-use in such cases, when we want to integrate the communication component in an existing application. In these cases it can easily happen that the structure of the existing application does not fit into the asynchronous, event-driven model. Then we would like handle communication “synchronously”.

In most cases, there is no activity in the communication; it is in a waiting state. As a client, our application is waiting for the response from the server. When our application acts as a server, it always has to listen for incoming client request. Our application seems to be only waiting for sending or receiving data. Therefore a suitable programming model had to be established, called asynchronous, event driven development.

The Symbian platform offers a special solution: it uses active object to avoid a continuous waiting state. An active object is a thread running parallel with the main application and generating communication requests and handling events that arrive as response for the requests. Only the platform can notify the active object about incoming outer events, but any object has access to the active object can cancel the request. The active object must implement a predefined interface, namely a method, which will be invoked by the OS (more precisely, by the scheduler) when a response arrives to any active request.

On the Symbian platform, an active object must be derived from the CActive class, and must implement the methods called RunL() and DoCancel(). The scheduler will call the method RunL(), when a response arrives for any request of the active object. It is important that only the OS (the scheduler) can call this method. The method DoCancel() is called directly by the public Cancel() method of the active object. It cancels all active request of the object.

A class hierarchy of the component can be seen in Figure 1. As a core, an active object must exist in the system, and it is inherited in the engine, in the reader, and in the writer objects. The active object either a part of the SDK of the platform, or can be implemented by the description. The Symbian platform has an implemented Active Object called CActive.

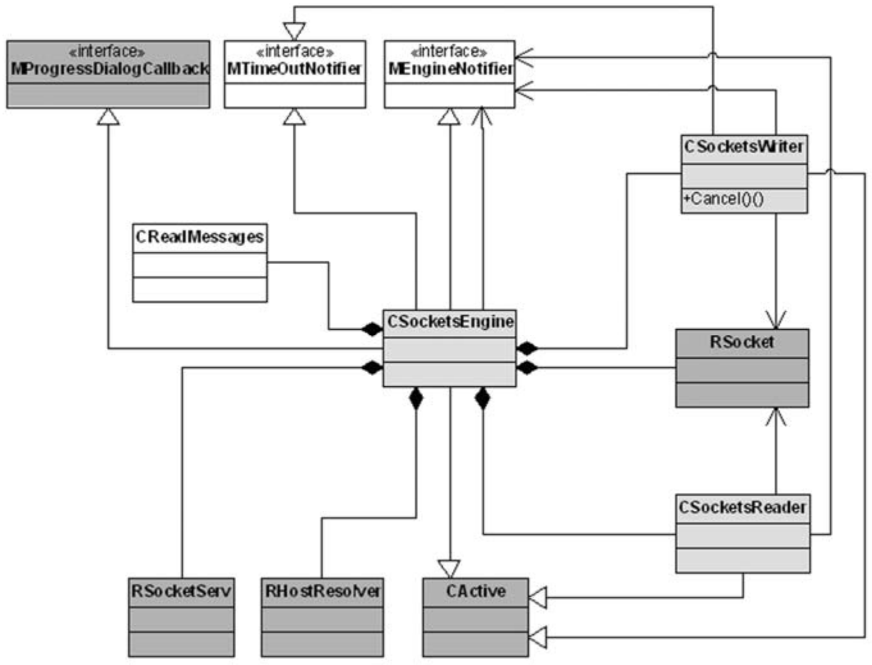

Figure 1
Structure of the communication component

The functions of the three classes mentioned above are the following:

- CSocketsEngine:
  - Creates socket on the server side and listens to incoming client calls.
  - Creates socket on the client side and connects to a server.
  - Handles all notifications from the reader and writer objects.
- CSocketsReader:
  - Reads continuously and asynchronously from the connected socket.
  - Notifies the engine object of the socket about incoming data.
- CSocketsWriter:
  - Writes data to the connected socket asynchronously.

Among all activities in the communication, connecting and reading from the socket are the two that mostly demonstrate the state waiting. A server must continuously listen for incoming client connections, and a client must continuously listen for incoming data to read. Therefore, the sequence diagrams of these activities are illustrated how they really work.

At first, we should take a closer look on at connection. When the server is started, it is in disconnected state. It makes initial configuration steps to select the communication technology should be used (IrDA, Bluetooth, TCP), specifies the port for the incoming connections, and finally opens a socket. Then it sets the active object active (says the OS then the object has a request) which notifies the engine about the incoming client connection.

Consider the scenario when a client tries to connect to a specified server on the specified port. Since this activity is asynchronous the platform (scheduler) will notify our active object, the communications engine by calling the method specified in the interface (in this case RunL()). When the connection process is closed successfully, the server and the client turn to connected state, and start a new continuous reading process.

A sequence diagram of the reading mechanism (normal, asynchronously) is illustrated in Figure 2. When the client and the server make a transition to the connected phase, they start a continuous reading. The reader object is realized as an active object. It sends out the read request, and “blocks” on the current socket waiting for incoming data. When data arrives, the platform notifies the reader object about it by calling the in the interface defined RunL() method. Then the reader object calls back the communication engine object on a predefined interface function ResponseReceived(). In this function, the engine notifies its owner class realizes the business logic of the application.

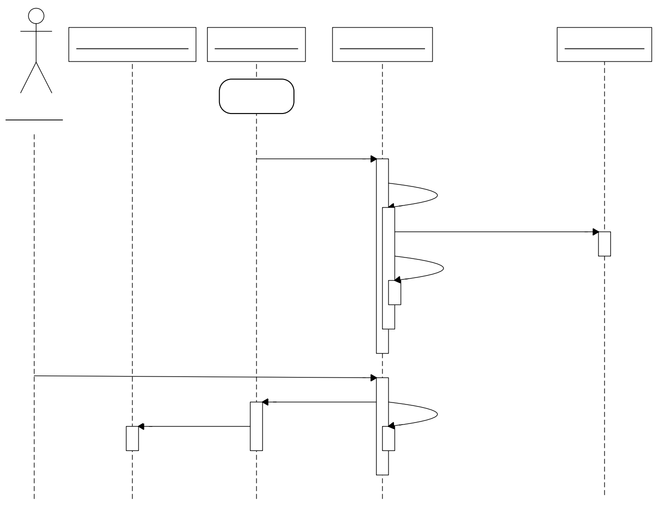

Figure 2

Reading from the socket

The sequence diagram of the other reading mode, the “synchronous” one is illustrated in Figure 3. By the point that data arrives on the socket it has the same functionality like the asynchronous one. When data arrives, the platform notifies the reader object about it by calling the in the interface defined RunL() method. Then the reader object calls back the communication engine object invoking a predefined interface function ResponseReceived(). In this function, the engine does not notify its owner class, only tries to store the arrived data. Therefore it has a FIFO queue of messages. The newly received data will be attached on the end of the list. Of course it is not an endless queue; therefore, it has a maximum length defined in the component. When it reaches this length, the engine object notifies its owner class about that error. When the main application reaches a point when it needs a received data, it calls the Read() method of the engine directly. The return value is either the oldest stored data if any, or NULL, representing that the message queue is empty. If there was a message in the queue, it will be given back to the owner object as a return value, and will be deleted from the front of the list.

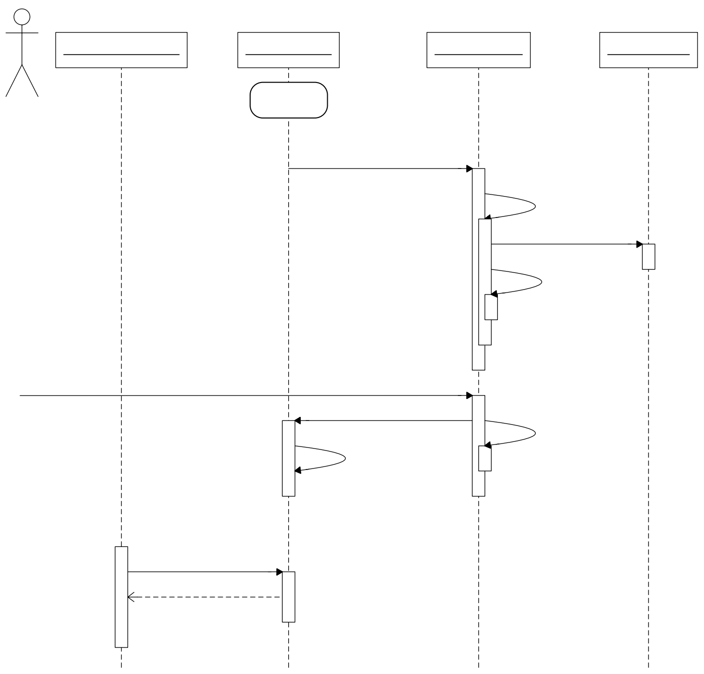

Figure 3
Reading from socket (synchronous)

Conclusion

In software development, it happens very often that the different applications have similar or same parts, like communication. When a developer needs to implement

Owner MEngineNotifier CSocketsEngine

EConnectec

the solution for the same sub-problem in every project, it will degrade the efficient of the whole development process, while:

- It takes a lot of time to create a real good communication engine.
- When there are no best practices for the recurring problems, very complex solutions can be realized depending on the experience of the developer.
- A complex solution could mean potential and typical errors, it is hard to debug, and could result an ineffective performance.

Therefore, it is necessary to have best practices and ready solutions for typical problems than can be easily reused or adopted on the target platform. It is also important that we spare time and resource, which can be invested to focus on the real problem of the specific application. When a concrete implementation is not ready for use, then design patterns can help us to design the right architecture and solutions for the application. They support solution for frequent problems in application development, like communication. With or without an example implementation, they are very useful and worth being followed, because time has proven their efficiency.

The need for such a communication library described above was verified by personal experience. Since Symbian Series60 platform has a wide range of APIs for using different device functionality, a multi-protocol communication is hard to implement, especially in each application. All of the mentioned communication technologies have a framework API to be used, but they differ from each other. The differences come up only in connection establishment, the rest of the communication are the same in each technique. In the initialization phase, depending on the used protocol we have to turn on the corresponding hardware component, set up a listening socket with appropriate properties, set up security (optional by TCP/IP and IrDA, but we must configure it for Bluetooth connection), do device discovery and finally connect to the remote server. During the configuration there are a lot of properties must be set up (especially by Bluetooth), which can have dedicated default values used by each connection. In most cases the developers do not want to customize all of these settings, they just want to use them in minimal line of source code. Therefore our library provides default values for these settings. Of course, when needed, their value can be set manually; the library has the support for that. We only have to define which communication method we want to use, and then connect to a remote device, transfer data, and finally disconnect.

We have found the implemented communication component very useful in software development on Symbian Series60 platform. With the help of this library we can easily use any kind of supported communication technology in our application with only a minimal number of new lines in our source code. Although the communication is an asynchronous operation, we can handle it as a synchronous one. With this new model it can fit easily the structure of our existing applications. By using this component, we can save the time of designing, implementing and testing the communication subsystem, where testing could be very hard, especially in mobile environment. Future work includes models based-support for the created communication library.

Acknowledgements

The fund of "Mobile Innovation Centre" has supported in part, the activities described in this paper.

References:

[1] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software (Addison-Wesley, 1994)

[2] Schmidt D. C., and Stephenson, P.: Experiences using design patterns to evolve system software across diverse OS platforms. In Proceedings of the Ninth European Conference on Object-Oriented Programming (Aarhus, Denmark), August 1995

[3] D. C. Schmidt: Using Design Patterns to Develop Reusable Object-Oriented Communication Software, COMMUNICATIONS OF THE ACM Vol. 38, No. 10, 1995, 65-74

[4] D. Schmidt, M. Stal, H. Rohnert, F. Buschmann: Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects, Volume 2 (John Wiley & Sons, 2000)

[5] D. C. Schmidt: “Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching,” in Pattern Languages of Program Design (Addison-Wesley, 1995)

[6] J. Vlissides, J. Coplien, N. Kerth: Pattern Languages of Program Design 2 (Addison-Wesley, 1996)

[7] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal: Pattern-Oriented Software Architecture - A System of Patterns (Wiley and Sons, 1996)

[8] Michael J. Jipping: Symbian OS Communications Programming (John Wiley & Sons, 2002)