7

SymTorrent and GridTorrent: Developing BitTorrent Clients on the Symbian Platform

Imre Kelényi
Budapest University of Technology and Economics, imre.kelenyi@aut.bme.hu

Bertalan Forstner
Budapest University of Technology and Economics, bertalan.forstner@aut.bme.hu

Abstract: This chapter aims to give an insight into how a complex peer-to-peer (P2P) application can be created on Symbian OS using C++. It focuses on BitTorent, which is one of today's most important P2P protocols. In addition to giving a short overview of the main concepts behind BitTorrent, two actual Symbian OS clients are introduced. SymTorrent and GridTorrent are the only Symbian OS BitTorrent clients to date. Through code snippets from these applications, many programming topics are covered, such as networking, sockets, and using the HTTP framework of Symbian OS. Since we do not go into the actual protocol implementation, but instead focus on the more general concepts, most of the topics covered here can be reused in any application using networking.

Keyword: BitTorrent; SymTorrent; GridTorrent

Contents

Mobile Peer to Peer (P2P) Edited by Frank H.P. Fitzek and Hassan Charaf

7.1 Introduction

At the point of writing, the BitTorrent protocol is one of the most popular alternative file transfer technologies on the Internet. In contrast to centralized solutions, such as HTTP or FTP, the BitTorrent protocol aims to transfer the data in a completely distributed way by dividing files into smaller pieces that can be retrieved from several locations. The other sources are BitTorrent users who have already downloaded all or part of the file. In return, the user's BitTorrent client may upload part of a file that has been previously downloaded. The key to scalable distribution is cooperation. Those who get a file use their own upload capacity to give the file to others at the same time. The greater the number of users downloading, the greater is the number of users uploading as well. This is the essence of BitTorrent, but more details are given in the next section.

Having this technology on mobile phones was not always possible earlier for several reasons. BitTorrent requires maintaining several network connections simultaneously and accessing multiple files at a time. However, with the introduction of powerful mobile hardware and open software platforms, which provide free development tools and enable third-party applications to be created, the time has come to bring peer-to-peer (P2P) file-sharing to mobile phones.

This chapter discusses two implementations of a BitTorrent client on Symbian OS, SymTorrent and GridTorrent. Both projects share the same code base.

SymTorrent was the first BitTorrent client for mobile phones. It was released in 2006 as an open-source project, and has since been downloaded more than half a million times. SymTorrent uses the standard BitTorrent protocol and so allows the downloading of any content shared with BitTorrent from the Internet in the same way as a desktop client.

GridTorrent is a specialization of SymTorrent that allows the users to form local clusters ('mini-networks' or 'grids', hence the name) and download files in a cooperative way, saving both bandwidth and energy.

The aim of this chapter is to give an insight into how complex P2P applications can be created on Symbian OS. Since SymTorrent and GridTorrent are quite large projects with thousands of lines of code and organized into several libraries, we cannot have a full coverage of the source code, but instead focus on the architecture, the key concepts, and the difficulties we faced during the development. This book is about programming, so, after giving an overview on the projects and how BitTorrent works, we jump right into writing code. We also discuss network interfaces, sockets, and Symbian's HTTP framework, so this chapter will be of interest to any developers interested in networking-based application and not just peer to peer.

BitTorrent is a peer-to-peer file-sharing protocol that was designed by Bram Cohen [1, 2]. After releasing the first fully functional version in 2003, it became an immediate success, and by 2004 the client software had been downloaded more than 10 million times. The key concepts are dividing the data into small pieces and transferring these between the participants in both directions. This means that, when one of the users has downloaded a piece, it can immediately start uploading it to another user. The pieces are not downloaded from a central source, but from the users themselves. Of course, an initial source is required, but, after enough users have started downloading, the number of requests to the original source reduces, to the point where it becomes redundant. BitTorrent also features an intuitive tit-for-tat mechanism that prevents free riding (downloading but never uploading) by giving

more bandwidth to those peers that upload data at a higher rate. Both the content provider and the user benefit from using BitTorrent, since the load on the servers is much lower, while the transfer speeds can be higher than with a central server. BitTorrent provides excellent redundancy, since it distributes data networkwide, limiting the problems caused by a central server going down.

In general, BitTorrent can be used to replace any centralized file transfer protocol, such as the FTP and HTTP protocols built into browsers, but at the moment BitTorrent clients are not as widespread.

To be able to understand the following parts of this chapter and the architecture of SymTorrent and GridTorrent, you must get familiar with the basics of the BitTorrent protocol.

When we refer to a *torrent*, we mean the file or files to be downloaded or shared. BitTorrent allows a single file or multiple files organized into directories (similar to a zip file) to be shared. After the creator of the torrent (the initial source of the data) has selected the files to be shared, there is no way of changing them. From this point in time, the torrent is a closed entity, and files cannot be added or removed. However, some BitTorrent clients allow only selected files to be downloaded from a multifile torrent. We often refer to the peers downloading and sharing a particular torrent as a *swarm*. Everybody who is in a swarm exchanges the pieces of a particular torrent.

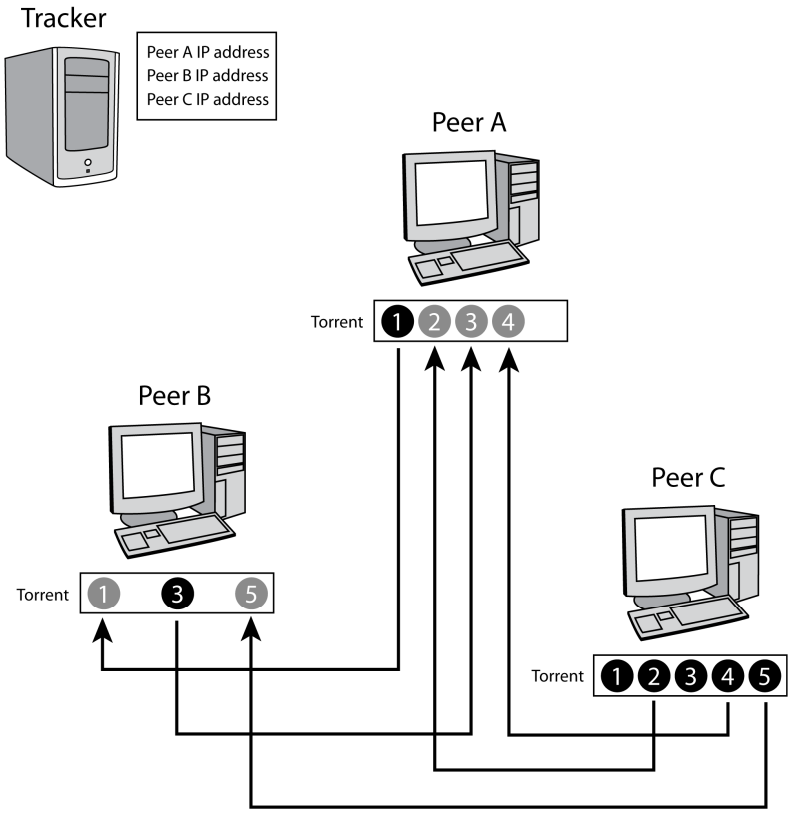

Figure 7.1 Transferring a torrent between three peers

To gain a better understanding of how data transfer works in BitTorrent, we will show a snapshot of the process of transferring a torrent between three peers. Figure 7.1 illustrates an example scenario. Peers are marked with letters A, B, and C. The bars under the peer icons show the status of the torrent. In this example, the torrent consists of five pieces. Real-life torrents usually have several hundreds or even thousands of pieces. The pieces are numbered

from 1 to 5. Black filled circles denote pieces that have been downloaded, while grey circles mark the pieces that are being downloaded. As you can see, one of the peers, peer C, has already downloaded the whole torrent and is acting as a *seeder*, a peer that only uploads. Peer A has one piece downloaded (piece 1) and is downloading three other pieces: one from peer B and two from peer C. Peer B has also finished one piece and is downloading two pieces, one from each of the other peers. As time progresses, peers A and B will have more and more pieces downloaded, which they can upload to each other, taking the load from peer C. This is just a simple example with three peers, but the same mechanism also works with thousands of clients. The inclusion of the tracker in the picture is to emphasize that it does not participate in the data transfer. The tracker does not have the torrent, and it only hosts the list of addresses of peers in the swarm.

Some of the most important terms of BitTorrent are as follows:

- *Tracker.* A central server whose task is to coordinate the peers that are participating in a swarm. The tracker itself does not share any data, it only maintains the list of peers that are downloading and sharing the torrent. Every torrent must have a tracker. Each time a peer starts downloading a torrent, it connects to the tracker, which provides it with the list of peers in the swarm. This procedure is referred to as *announcing*, for the peer also announces its own address to the tracker. The tracker is a fundamental component of BitTorrent: if it is not available, new peers cannot join the swarm. The peers communicate with the tracker via standard HTTP GET requests. It should be noted that a tracker can host several torrents, and a torrent is often registered in several trackers in order to achieve some level of redundancy. Although the newest version of BitTorrent supports *trackerless* torrents [3], which store and retrieve peer addresses from a distributed hash table (DHT), this is mainly used only if the tracker is unavailable. Hence, in this chapter we focus on the standard tracker solution only.
- *Torrent file.* A binary[1] file that contains all the required information to download a torrent (join the swarm). It contains a link (a URL) to the tracker and the list of files in the torrent, along with the hash values of the pieces of the torrent. These SHA-1 hash values allow the integrity of the downloaded data to be checked. Other metadata, such as the creator of the torrent, can also be added when the torrent file is created. As mentioned earlier, a torrent cannot be changed after it has been created. In practice, this is achieved by using the hash of the torrent file as the ID for the torrent. This ID is used in several protocol messages, and thus it cannot be changed. Since changing the torrent file would change its ID as well, this is not possible without violating the protocol. It is important to understand that the *torrent file* is not synonymous with the *torrent*. The former is just a reference to the swarm, while the latter is the swarm itself.
- *Peers (leechers and seeders).* Users (computer, mobile phones, etc.) running an instance of a BitTorrent client. Peers are the source of data, and they are also the downloaders. If there are no peers available, the torrent cannot be downloaded. The protocol that peers use to communicate with each other is referred to as the *peer wire*. It defines a set of messages, such as REQUEST (requests a block of a piece) or PIECE (sends a block of a piece). The peers are often divided into two categories:
  - *Leechers.* Peers that have not downloaded the full torrent yet. They are both uploading and downloading.
  - *Seeders.* Peers that have the full torrent. They are not downloading any more, but keep uploading the torrent. When the torrent is created, an initial seeder is required to host the shared data until it is spread in the network. Seeders have a positive effect on the overall available bandwidth of the torrent.

[1] The data in torrent files is in a format referred to as *bencode*.

To summarize how the protocol works, here are the steps for creating a new torrent and starting to share it:

1. The files that will be shared are selected.
2. Using a special application, the *torrent maker*, the torrent file, is created. The files that are shared and the address of the tracker must be given.
3. The torrent is registered in the tracker whose address was encoded into the torrent file.
4. The initial seeder starts sharing the torrent by announcing to the tracker.

After the initial source has started sharing the torrent, any peer can join the swarm, provided it possesses the torrent file. Without the torrent file, theoretically, it is not possible to join a torrent, since it is the only source from where the torrent's ID and the address of the tracker can be obtained. After the peer has announced to the tracker and received the list of some other peers, it can start establishing connections to them and transfer pieces of the torrent.

7.2 SymTorrent

SymTorrent is a complete BitTorrent client for Symbian OS. Currently, it is released for the S60 3rd edition platform, but most of the code is platform independent, and only the UI layer is specific to S60. SymTorrent features a multiview user interface, allows multiple torrents to be downloaded at a time, and can resume torrents after exiting the application. At the point of writing, torrents must be added manually in SymTorrent (via the 'Add torrent' dialog). Compared with a PC client, SymTorrent lacks some of the more advanced features, such as peer exchange, NAT traversal, scheduling, etc. However, in terms of downloading, SymTorrent performs reasonably well. The source code is freely available under the terms of the GNU General Public License at http://symtorrent.aut.bme.hu.

Figure 7.2 A screenshot of the main view of SymTorrent, showing two loaded torrents: the first is paused, the second is being downloaded from six peers

SymTorrent was written entirely in Symbian C++ using the S60 3rd edition SDK MR. It does not use the recently introduced Open C/C++ framework, nor any other add-on libraries. In terms of the architecture, when the first public version was released, SymTorrent consisted of a single executable only. Later, as the application was developed and the source code became larger and larger, the code was divided into several components. After putting the UI layer into a separate DLL, we also started dividing the 'engine' part into several libraries that can be maintained more easily. At this point, SymTorrent consists of the following subprojects:

- *SymTorrent.* The name might be a bit misleading, but this is only the UI-dependent part of SymTorrent. This project is responsible for creating the different UI views, initializing the engine, and processing the user inputs. Practically, this project creates the executable that starts when the application is selected in the phone's menu.
- *SymTorrentEngine.* This is the essence of SymTorrent that does all the non-UI-dependent tasks. The engine is responsible for implementing the BitTorrent protocol, including parsing torrent files, announcing to the tracker, and establishing peer-wire session with the other peers. Technically, SymTorrentEngine is a DLL. Any third-party application can use it, provided that it implements a couple of predefined interfaces.
- *LibBencode.* A DLL implementing BitTorrent bencode decoding and encoding. Bencode is the binary format of torrent files and the tracker requests.
- *KiNetwork.* Since SymTorrent depends heavily on networking, we decided to create a separate DLL that is responsible for almost all networking tasks, such as initializing the network interfaces, handling sockets, and accepting incoming connections. KiNetwork provides base classes for objects using UDP, TCP, or even Bluetooth sockets. These classes make Symbian socket programming much easier by providing simple methods for writing to the socket and receiving data. KiNetwork can also be used to initialize the network interface of HTTP sessions.
- *KiLogger.* A simple set of classes that allow the creation and manipulation of log files. Generally, log files are used for debugging purposes. Many parts of the code of SymTorrent contain logging calls, which write debugging information into a text file.
- *SymTracker.* Since SymTorrent was started as a research project, we wanted to experiment with several scenarios, including having a separate tracker on the phone itself. SymTracker is a very simple tracker that can host a list of peers and provide them to the announcing peers. However, it also has a built-in torrent maker function, which allows torrents to be created and hosted in a few steps on the device. Since SymTracker is an optional component and not a fundamental part of SymTorrent, it can be ignored if you are not interested in hosting a separate tracker on the device.

When you build SymTorrent using the SDK's development tools, all of these projects are compiled, linked, and, in the final phase, if you build for the device, combined into an installable SIS file. In this chapter we mainly focus on certain parts of SymTorrentEngine and KiNetwork.

7.3 GridTorrent

GridTorrent uses the same engine as SymTorrent. The newest versions of SymTorrent and GridTorrent both build on top of SymTorrentEngine, which also contains the application logic of GridTorrent. The differences are only in the main UI projects. GridTorrent has a different download status view, with statistics on the local connections. It also shows a special status bar that displays which pieces have been downloaded from the local network.

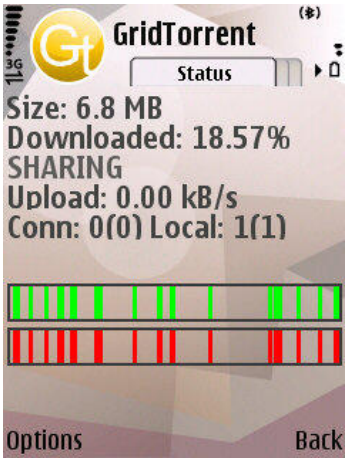

Figure 7.3 A screenshot from GridTorrent, showing the status of a torrent being transferred from one local peer

Otherwise, GridTorrent is very similar to SymTorrent. It is a BitTorrent client that enables local cooperation in downloading torrents. This means that users can form small local networks (clusters, grids) that are connected via WLAN or Bluetooth and cooperate to download torrents more efficiently. The peers in the local network download pieces both from each other and from peers on the Internet. Downloading via the local links can be faster and more efficient; thus, downloading pieces from local peers is preferred. The locally connected peers share extra information on their status with each other. The goal is to minimize data traffic with peers connected over the long-range links and obtain as much data as possible from the local cluster. This is how cooperating GridTorrent peers conserve both energy and data traffic.

The topology of a network with GridTorrent peers is illustrated in Figure 7.4. The peers using GridTorrent, which are marked with the phone icon, are connected over short-range links, typically over Bluetooth. They form the local cluster, which is marked with a cloud. Besides the locally connected peers, GridTorrent clients also establish connections with peers on the Internet, over a long-range network interface, which can be HSDPA, GRPS, or even WLAN.

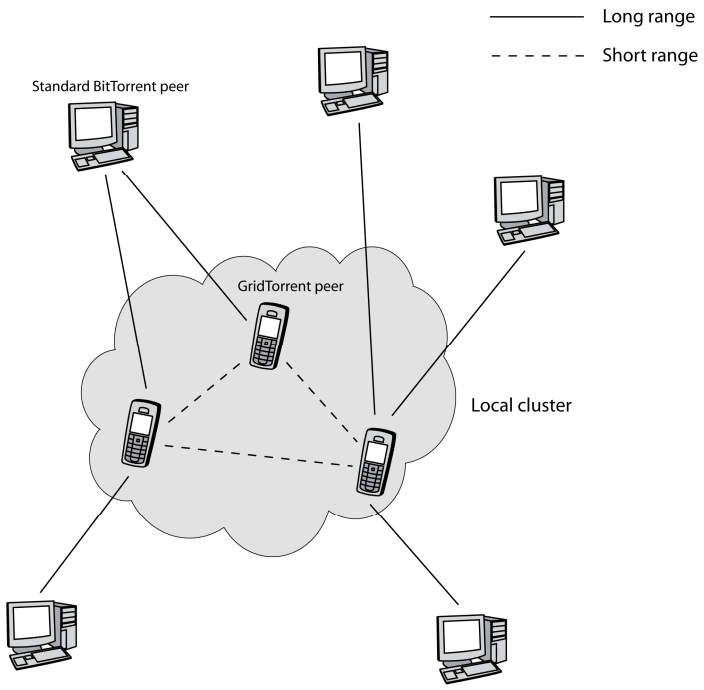

Figure 7.4 Topology of a GridTorrent-based network

GridTorrent can be used as a standard BitTorrent client, and torrents can be added and downloaded from the Internet without using any of the added features. However, there are a couple of new options in the menus that enable us to establish local connections between the devices. By selecting 'Start local listening', a phone starts listening via the selected network interface, which can be Bluetooth or WLAN. If we want to make a connection to a local device that is already listening, this can be done by selecting the 'Add local peer' option. Here, the IP address of the device can be given, or, in case of Bluetooth, a device discovery dialog is shown. After the local links have been established, the devices download pieces from the Internet and from each other in a cooperative way. Pieces that are available locally are downloaded from the local peers.

At the point of writing, GridTorrent has not been released to the public, but we are planning to make it open source. Generally speaking, GridTorrent is currently a research project, but we think that any developer can benefit from our experiments with creating the application.

7.4 Developing a BitTorrent Client

Designing and implementing a P2P client on Symbian OS is a demanding task. The application logic is complex and the programmer must be familiar with many advanced features of the platform. This chapter aims to give you some tips, ideas, and code fragments that will help you take the first steps. In accordance with the topic of this book, we focus on the networking aspects of the application. We do not go into the implementation of the BitTorrent protocol, and most of the topics covered here can be reused in any application using networking. The code snippets are based on the engine of SymTorrent and GridTorrent; however, they are not simply copied and pasted code. The classes are greatly simplified. For example, we have removed parts of the code that are considered less significant, and, to simplify the code, we have left out `NewL()` and `NewLC()` methods and some other definitions.

Nevertheless, you can always refer to the full open-source version of SymTorrent and GridTorrent [4]. We try to point out the projects in which you can find the associated source

files. From now on we will use SymTorrent when we refer to the complete mobile application. The header files for the Symbian-based classes used can be found by searching the help file of the SDK. All of the source code in this chapter was tested with the S60 SDK 3rd edition MR (Maintenance Release).

Figure 7.5 shows the simplified class diagram of the engine of SymTorrent. It does not include the user interface classes, and several lower-level classes are also omitted.

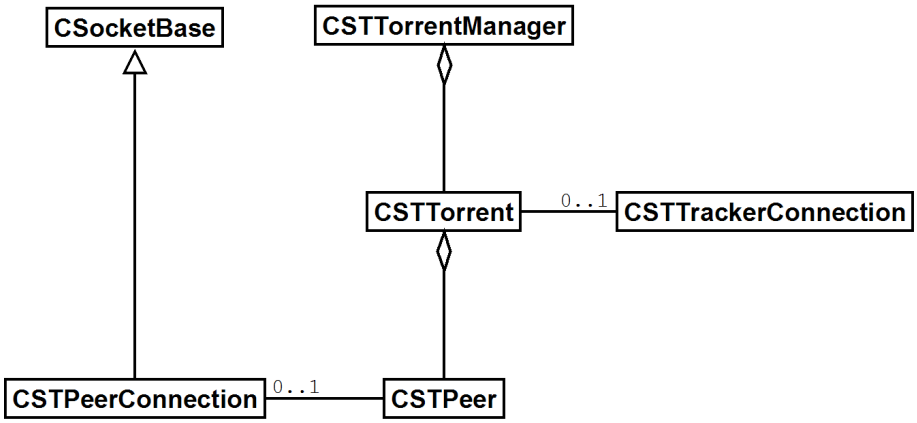

Figure 7.5 The simplified class diagram of the engine of SymTorrent

7.4.1 Creating the Network Manager

Before moving on to the BitTorrent-related classes, we are going to implement a simple networking framework that makes network and socket programming simpler. This is the lowest level of the application, where network connections are created and sockets are read and written; thus, these code snippets can be utilized in almost any application that uses networking. The full source code can be found in the KiNetwork subproject of SymTorrent. Figurre 7.6 shows the classes of the networking framework that we are going to implement.

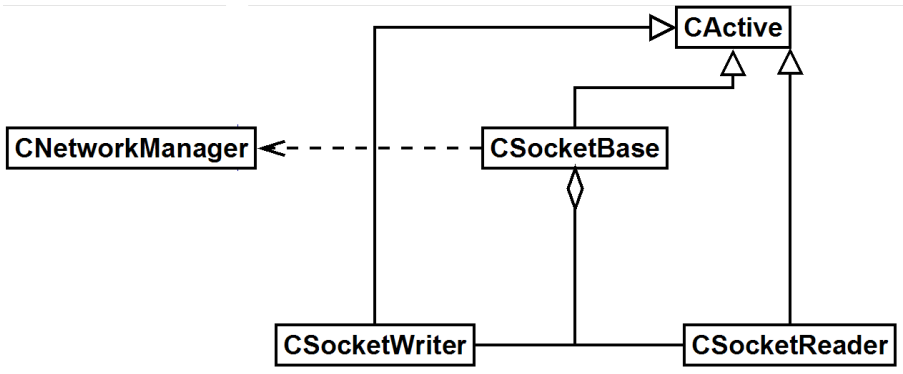

Figure 7.6 The simplified class diagram of the networking layer

The network manager is responsible for starting and maintaining the active network connection, providing a socket server session and listening for incoming connections. These are implemented by a single class referred to as `CNetworkManager`. This is a singleton class, which means that only one instance of it can exist at any time, and this instance can be accessed globally. Traditionally, singletons are implemented using static class members; however, in Symbian OS, it is not preferred to use global writable static data in DLLs [5]. Instead, thread-local storage (TLS) is used. TLS is a single machine word of static memory

whose scope is the thread in which the code of the DLL is running. In a DLL that uses multiple singleton classes, the TLS has to be set to some container class that manages the singletons. In this example, however, we are going to use only one singleton class per DLL. Thus, the TLS can be safely set to the address of this singleton. The declaration of the class is as follows:

```
class CNetworkManager : public CBase
  {
public:
  static CNetworkManager* Instance(); // Singleton access
  static void InitializeL(); // Singleton initialization
  static void Free(); // Singleton cleenaup

  ~CNetworkManager();

  void StartNetworkConnectionL(
    MNetworkConnectionStarterObserver* aObserver = NULL);

  TBool IsNetworkConnectionStarted() { return iNetworkConnectionStarted; }

  void StartListeningL(TUint aPort, MSocketListenerObserver* aObsever);
  void StopListening();

  RSocketServ& SocketServ() { return iSocketServer; }
  RConnection& NetworkConnection() { return iConnection; }

private:
  CNetworkManager() : CCoeStatic(KUidNetworkManagerSingleton) {}
  void ConstructL();
  void GetIapNamesAndIdsL(RArray<TUint32>& aIds, CDesC16Array& aNames);
  TUint32 QueryIapIdL();

private:
  TBool iNetworkConnectionStarted;
  RConnection iConnection;
  RSocketServ iSocketServer;
  CSocketListener* iSocketListener;
  TInt iReferenceCount;
  };
```

Before any access can be made to `CNetworkManager`, the static method `InitializeL()` must be called. It creates the singleton instance if it is not initialized yet, and increments a reference counter, which is used during clean-up to ensure that the object is not cleaned up while it is still in use:

```
void CNetworkManager::InitializeL()
  {
  // Get TLS
  CNetworkManager* instance = (CKiLogManager*)Dll::Tls();

    if (instance == 0)
      {
      instance = new (ELeave) CNetworkManager();
      CleanupStack::PushL(instance);
      instance->ConstructL();
      CleanupStack::Pop();

      Dll::SetTls(instance); // Set TLS to the singleton
      }

    instance->iReferenceCount++;
}
```

Every call to `InitializeL()` must be paired with a call to the static method `Free()`, which cleans up the object. Reference counting enables the singleton to be used in a shared DLL. The reference counter is decreased each time `Free()` is called. When the counter reaches

zero, the singleton is cleaned up. For debug purposes, a panic is raised if the singleton is freed up too many times:

```
void CNetworkManager::Free()
  {
  CNetworkManager* instance = (CNetworkManager*)Dll::Tls();

  if (instance)
    {
    instance->iReferenceCount--;

    if (instance->iReferenceCount == 0)
      {
      delete (CNetworkManager*)Dll::Tls();
      Dll::SetTls(NULL);
      }
    }
  else
    User::Panic(KLitNetworkManagerPanic, ENetworkManagerFreedUpTooManyTimes);
}
```

Accessing the singleton is performed by calling the static `Instance()` method, which does nothing else but return the content of the TLS:

```
CNetworkManager* CNetworkManager::Instance()
  {
  return (CNetworkManager*)Dll::Tls();
  }
```

To simplify accessing the network manager, we define the inline function `NetMgr()`:

```
inline CNetworkManager* NetMgr()
  {
  return                                                CNetworkManager::Instance();
  }
```

The second-phase constructor of `CNetworkManager` opens a session to the socket server:

```
void CNetworkManager::ConstructL()
  {
  User::LeaveIfError(iSocketServer.Connect(255));
  }
```

The destructor releases the owned resources, namely the socket server handle, the network connection handle, and the socket listener, if it has been initialized:

```
CNetworkManager::~CNetworkManager()
  {
  iSocketServer.Close();
  iConnection.Close();
  delete iSocketListener;
  }
```

7.4.2 Network Connections

We are going to create a P2P application, which means that our program needs access to the Internet. To do so, we must establish a network connection. By network connection we mean the actual network interface, for example WLAN or GPRS. By default, the framework pops up a network connection selection dialog when the network is first accessed, and automatically starts the selected network connection. However, this pop-up can sometimes be annoying, especially if it pops up multiple times. In order to provide a better user experience,

the network selection dialog can be avoided by setting up the desired connection directly. Another plus is that, by directly programming the connection, it can be saved and reloaded after the application restarts. Under Symbian OS, each network connection is associated with an access point. An access point is an entry in the communications database. It has several fields, including a name and a unique ID. To start a connection, we must first obtain its ID. In the following example we are going to query all available access points from the framework and display them in a dialog for the user. On the S60 platform, the access point engine can also be used via the `CApSelect` class to access the available access points. However, in this example we are going to extract the access points directly from the communications database. The full process of querying the access point and starting the network connection is illustrated in Figure 7.7

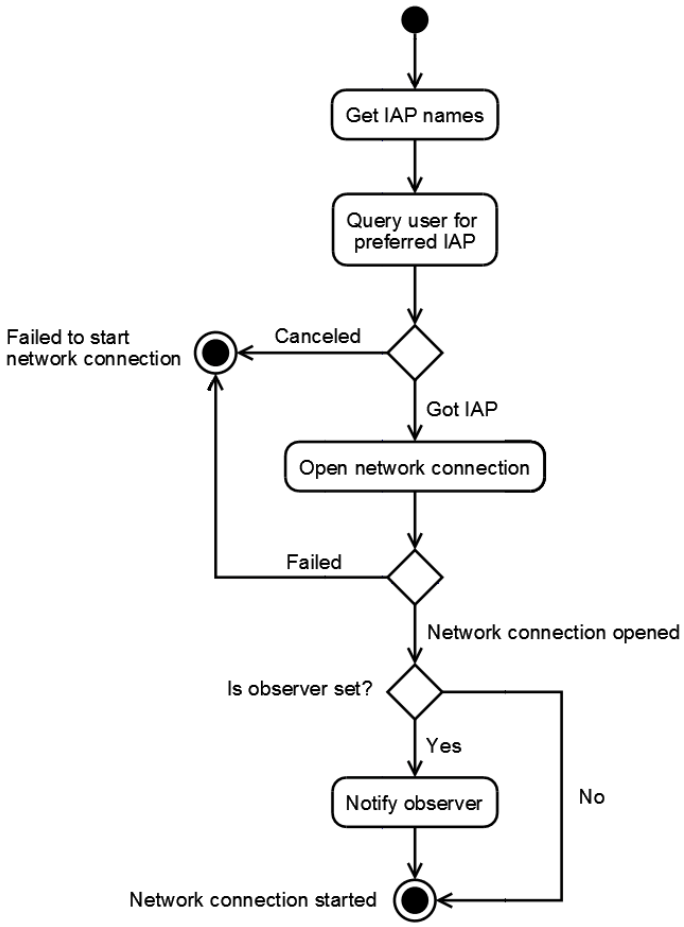

Figure 7.7 Activity diagram of starting the network connection

The first step is to get the names and IDs of the available access points. To do so, we need to access the Internet access point (IAP) table of the communications database and get all of its records. The following method extracts the access point fields into the passed arrays:

```
void CNetworkManager::GetIapNamesAndIdsL(RArray<TUint32>& aIds,
  CDesC16Array& aNames )
  {
  // Open COMM database
  CCommsDatabase* commsDb = CCommsDatabase::NewL();
  CleanupStack::PushL(commsDb);

  // Get the table with the access points
  CCommsDbTableView* view = commsDb->OpenTableLC( TPtrC(IAP) );
```

```
  TInt res = view->GotoFirstRecord();

  // Read all the access points
  while (res != KErrNotFound)
    {
    User::LeaveIfError(res);

    TBuf<KCommsDbSvrMaxFieldLength> name;
    TUint32 id;
    view->ReadTextL(TPtrC(COMMDB_NAME), name);
    view->ReadUintL(TPtrC(COMMDB_ID), id);

    aIds.Insert(id, 0);
    aNames.InsertL(0, name);

    res = view->GotoNextRecord();
    }

  CleanupStack::PopAndDestroy(2, commsDb); // view, commsDb
}
```

With the name and ID of the access points in hand, we are able to display a dialog for the user. `QueryIapIdL()` creates a pop-up list dialog, populates it with the access points, displays it to the user, and returns the ID of the selected item. This dialog is very similar to the one that is automatically displayed by the OS, but it lacks the small icons:

```
TUint32 CNetworkManager::QueryIapIdL()
  {
  RArray<TUint32> idArray;
  CleanupClosePushL(idArray);
  CDesCArrayFlat* namesArray = new (ELeave) CDesCArrayFlat(5);
  CleanupStack::PushL(namesArray);

  // Get access points
  GetIapNamesAndIdsL(idArray, *namesArray);

  // Create the popup list
  CEikTextListBox* list = new (ELeave) CAknSinglePopupMenuStyleListBox;
  CleanupStack::PushL(list);
  CAknPopupList* popupList = CAknPopupList::NewL(list,
    R_AVKON_SOFTKEYS_OK_CANCEL,AknPopupLayouts::EMenuWindow);
  CleanupStack::PushL(popupList);

  // Initialize the listbox.
  list->ConstructL(popupList, CEikListBox::ELeftDownInViewRect);
  list->CreateScrollBarFrameL(ETrue);
  list->ScrollBarFrame()->SetScrollBarVisibilityL(CEikScrollBarFrame::EOff,
                                                  CEikScrollBarFrame::EAuto);

  // Set list items
  CTextListBoxModel* model = list->Model();
  model->SetItemTextArray(namesArray);
  model->SetOwnershipType(ELbmDoesNotOwnItemArray);

  // Set title
  popupList->SetTitleL(_L("Select connection"));

  // Show popup list
  TBool changed = popupList->ExecuteLD(); // shows the dialog
  CleanupStack::Pop(); // popuplist

  TInt iapId = 0;
  if (changed)
    iapId = (TUint32)idArray[list->CurrentItemIndex()];

  CleanupStack::PopAndDestroy(3, idArray); // list, namesArray, idArray
  return iapId;
  }
```

The next step is to define the method that actually starts the network connection. Like many networking function calls, starting the connection is an asynchronous process. Thus, it should

be called by an active object [6] so that the thread of the application is not blocked. In this example, however, we use a synchronous `WaitForRequest()` call to make the code more readable. Nevertheless, we show an observer class that could be used if we implemented the asynchronous version. The caller can pass an implementation of the observer whose `NetworkConnectionStartedL()` method is called when starting of the network connection is completed or fails:

```
class MNetworkConnectionStarterObserver
  {
public:
  virtual void NetworkConnectionStartedL(TInt aResult,
    RConnection& aConnection) = 0;
};
```

The `StartNetworkConnectionL()` method calls `QueryIapIdL()` to obtain the access point ID from the user. The network connection is encapsulated by the `RConnection` class, which must be opened before it can be used. Starting the connection is done by configuring the fields of a `TCommDbConnPref` object with the acquired access point and passing it to the `Start()` method of the connection. Since we use a synchronous `WaitForRequest()` call, the thread is blocked at this point until the connection has been started or the operation fails. We can obtain the result from the passed `TRequestStatus` object. Finally, we signal the observer if it is available:

```
void CNetworkManager::StartNetworkConnectionL(
  MNetworkConnectionStarterObserver* aObserver)
  {
  TUint32 iapId = QueryIapIdL(); // Query the access point ID

  if (iapId)
    {
    TCommDbConnPref prefs;
    prefs.SetIapId(iapId);
    prefs.SetDirection(ECommDbConnectionDirectionOutgoing);
    prefs.SetDialogPreference(ECommDbDialogPrefDoNotPrompt);
    prefs.SetBearerSet(KCommDbBearerUnknown);

    User::LeaveIfError(iConnection.Open(iSocketServer));

    TRequestStatus status;
    iConnection.Start(prefs, status); // Start the connection
    User::WaitForRequest(status); // ActiveObject should be used

    if (status.Int() == KErrNone)
      iNetworkConnectionStarted = ETrue;

    if (aObserver)
      aObserver->NetworkConnectionStartedL(status.Int(), iConnection);
    }
  }
```

After the network connection has been started, it can be used to initialize sockets, host resolvers, or HTTP sessions. All of these classes have a method that can be used to attach an `RConnection` instance that encapsulates the started network connection.

7.4.3 Listening for Incoming Connections

In BitTorrent, peers communicate with each other over TCP/IP connections. To accept incoming connections, we need to start listening on a port. If an incoming connection is accepted on the given port, the framework gives us a configured socket that can be used to communicate with the other party. By its nature, listening is an asynchronous process. Symbian OS allows us to do listening synchronously, but it does not make too much sense,

since this would block the entire thread. Hence, we show you how to implement listening asynchronously by using active objects.

First of all, we create a callback method encapsulated in an observer class `MSocketListenerObserver`. This class is responsible for notifying the client when an incoming connection is accepted:

```
class MSocketListenerObserver
  {
public:
  // Called when an incoming connection is accepted, the ownership of the socket
  // is passed to the called object.
  virtual void AcceptSocketL(RSocket& aSocket) = 0;
  };
```

We implement listening in the active object `CSocketListener`. Listening is started by calling `StartListening()` and specifying a port and an observer. The private fields of the class include two socket handles: `iSocketListener` is the socket that actively listens and `iBlankSocket` is the socket that encapsulates the next incoming connection:

```
class CSocketListener : public CActive
  {
public:
  CSocketListener() : CActive(EPriorityStandard) {}
  ~CSocketListener();

  TInt StartListening(TUint aPort, MSocketListenerObserver* aObserver);
  void StopListening();

private: // from CActive
  void RunL();
  void RunError();
  void DoCancel();

private:
  MSocketListenerObserver* iObserver;
  RSocket iSocketListener;
  RSocket iBlankSocket;
  };
```

`StartListening()` begins with checking whether the object has been activated. Before we can start listening, we must make sure that the network connection has been started. If the connection is offline, then we start it through the network manager. Since we implemented `StartNetworkConnectionL()` as a synchronous method, we do not need to pass an observer and wait for the result.

To start listening, the listening socket must be opened first. It is important to pass the started network connection as the fourth parameter of the `Open()` method. Otherwise, the automatic access point selection dialog is displayed. Then we must specify the port on which the socket listens by binding an arbitrary network address with the selected port to the socket. Calling `Listen()` starts listening, but incoming connections are not accepted until `Accept()` is called. `Accept()` is an asynchronous operation. Its first parameter is the blank socket that will be attached to the incoming connection. The asynchronous operation is completed (and the active object event handler method `RunL()` is executed) when an incoming connection arrives:

```
TInt CSocketListener::StartListening(TUint aPort,
  MSocketListenerObserver* aObserver)
  {
  if (IsActive()) User::Panic(…);
  iObserver = aObserver;

  if (!NetMgr()->IsNetworkConnectionStarted())
    {
```

```
    TRAPD(err, NetMgr()->StartNetworkConnectionL());
    if (err != KErrNone)
      return err;
    }

  TInt err = iSocketListener.Open(NetMgr()->SocketServ(), KAfInet,
    KSockStream, KProtocolInetTcp, NetMgr()->NetworkConnection()));

  if (err != KErrNone)
    return err;

  TInetAddr addr;
  addr.SetPort(aPort);

  err = iSocketListener.Bind(addr);
  if (err != KErrNone)
    return err;

  err = iSocketListener.Listen(5);
  if (err != KErrNone)
    return err;

  iBlankSocket.Close();
  iBlankSocket.Open(NetMgr()->SocketServ());
  iSocketListener.Accept(iBlankSocket, iStatus);
  SetActive();

  return KErrNone;
  }
```

We also provide a method to stop the listening process. Two calls are performed: the active object and the asynchronous `Accept()` request are cancelled by calling `Cancel()`, and the resources of the listening socket are released by calling the `Close()` method of the socket handle:

```
void CSocketListener::StopListening()
  {
  Cancel();
  iSocketListener.Close();
  }
```

The last method we need to implement in the socket listener class is the event handler method of the active object. `RunL()` is called when the `Accept()` request is completed. The completion of the request does not necessarily mean that it has been successful. Thus, we need to check the result by accessing the `iStatus` member variable of the active object. If it is set to `KErrNone`, then accepting an incoming connection has been successful, and `iBlankSocket` is attached to it. In this case, we pass the socket to the observer. It should be noted that the observer must take the ownership of the new socket. After passing the socket, a new `Accept()` request can be issued to continue listening. This is done by reinitializing the blank socket, issuing the request, and setting the object to the active state. If the `Accept()` request fails, we stop the listening process:

```
void CSocketListener::RunL()
  {
  if (iStatus.Int() == KErrNone)
    {
    if (iObserver)
      iObserver->AcceptSocketL(iBlankSocket);
    else
      iBlankSocket.Close();

    // Initialize a new blank socket
    iBlankSocket = RSocket();
    iBlankSocket.Open(NetMgr()->SocketServ());
    iSocketListener.Accept(iBlankSocket, iStatus);
    SetActive();
    }
```

```
  else
    StopListening();
  }
```

We also need to implement `DoCancel()`, the method that cancels the asynchronous request. In our case, it calls the `CancelAll()` method of the listener socket, which cancels the active `Accept()` request:

```
void CSocketListener::DoCancel()
  {
  iSocketListener.CancelAll();
  }
```

The destructor cancels the active object and closes both owned socket handles:

```
CSocketListener::~CSocketListener()
  {
  Cancel();
  iBlankSocket.Close();
  iSocketListener.Close();
  }
```

Now that we have defined `CSocketListener`, we should also add the listening methods to the network manager. In this way, listening for incoming connections can be started by calling a method of the easily accessible singleton network manager. Listening is started by calling `StartListeningL()`, which creates a new instance of the socket listener and starts listening. Stopping listening is performed by calling `StopListeningL()`, which deletes the socket listener instance:

```
void CNetworkManager::StartListeningL(TUint aPort,
  MSocketListenerObserver* aObserver)
  {
  if (iSocketListener == NULL)
    iSocketListener = new (ELeave) CSocketListener;

  if (!iSocketListener->IsActive())
    User::LeaveIfError(iSocketListener->StartListeningL(aPort, aObserver));
  }

void CNetworkManager::StopListening()
  {
  delete iSocketListener;
  iSocketListener = NULL;
  }
```

7.4.4 Sending Data Via Sockets

Communicating with the peers is performed via sockets. Reading and writing sockets must be handled asynchronously so that an active operation does not block the entire application. We are going to create a generic socket class that encapsulates a socket handle and can be used to write to the socket and get notifications when data is received from the connected peer. This class is referred to as *socket base* (`CSocketBase`). Since both sending and receiving are handled asynchronously, both of these functions require separate active objects (`CSocketWriter` and `CSocketReader`).

The first class we are going to implement is `CSocketWriter`, an active object that encapsulates an asynchronous write request to the socket server. The first-phase constructor takes the owner socket class as a reference (`CSocketBase`); this class will be implemented later. The socket writer does not open a socket or make any connections. Its purpose is to send data via an already initialized and connected socket. The socket writer can only have one active write operation. This means that, if we want to send data when there is already an

active request, then we must store the data in a buffer and send it later. This way, an arbitrary number of requests can be issued to the socket writer, and the write operations will be performed in a sequential order:

```
class CSocketWriter : public CActive
  {
public:
  CSocketWriter(CSocketBase& aSocketBase)
    : CActive(EPriorityStandard),
     iSocket(aSocketBase.Socket()), iSocketBase(aSocketBase) {}

  void ConstructL();
  ~CSocketWriter();
  void WriteL(const TDesC8& aBuf);

private:
  void IssueWrite();

private: // from CActive
  void RunL();
  void DoCancel();

private:
  RSocket& iSocket;
  CSocketBase&        iSocketBase;
  CBufSeg*            iLongBuffer;
  RBuf8               iShortBuffer;
  };
```

In the second-phase constructor we add the active object to the active scheduler and then initiate the two data buffers that are used for sending the data. A shorter buffer is used for the actual write requests, and a longer buffer is used to queue the writable data until a new request can be issued:

```
void CSocketWriter::ConstructL()
  {
  CActiveScheduler::Add(this);
  iShortBuffer.CreateL(16384); // 16 KB

  iLongBuffer = CBufSeg::NewL(256);  // 256 byte for the granularity
  }
```

The destructor frees up the resources and cancels the active object:

```
CSocketWriter::~CSocketWriter()
  {
  Cancel();
  iShortBuffer.Close();
  delete iLongBuffer;
  }
```

The `WriteL()` method puts the writable data into the write buffer and issues a new write operation if the active object is not active. If it is already in the active state, then the data will be sent after the current write request has been completed:

```
void CSocketWriter::WriteL(const TDesC8& aBuf)
  {
  iLongBuffer->InsertL(iLongBuffer->Size(), aBuf);
  if (!IsActive()) IssueWrite();
  }
```

Issuing a new socket write request is performed by filling up the short transfer buffer and calling the asynchronous write operation. The data is read from the long buffer and is deleted immediately from there:

```
void CSocketWriter::IssueWrite()
  {
  if (iLongBuffer->Size() < iShortBuffer.MaxLength())
    iLongBuffer->Read(0, iShortBuffer, iLongBuffer->Size());
  else
    iLongBuffer->Read(0, iShortBuffer);

  iLongBuffer->Delete(0, iShortBuffer.Length());

  iSocket.Write(iShortBuffer, iStatus);
  SetActive();
  }
```

Cancelling the socket writer is performed by calling the `CancelWrite()` method of the socket handle:

```
void CSocketWriter::DoCancel()
  {
  iSocket.CancelWrite();
  }
```

In the event handler method, we get the result of the last write request. If it has been successful, we check whether there is data in the send buffer. If the buffer is not empty, a new write request is issued. If the write request fails, then the `HandleWriteErrorL()` method of the owner socket base is called. This is a virtual method that can be implemented by classes derived from `CSocketBase()` being notified when writing to the socket fails:

```
void CSocketWriter::RunL()
  {
  switch (iStatus.Int())
    {
    case KErrNone: // Writing to socket has been completed
      if (iLongBuffer->Size() > 0) IssueWrite();
    break;

    default: // Write error
      iSocketBase.HandleWriteErrorL();
    break;
    }
  }
```

7.4.5 Receiving Data from Sockets

Similarly to writing data to a socket, reading is also an asynchronous process and must be handled via an active object. We are going to show you how to implement the socket reader class `CSocketReader` which can actively read incoming data from a connected socket. This class will also be owned by the base socket class `CSocketBase`; thus, we pass a reference of the owner to the constructor. A reference to a byte buffer is also passed. The received data will be put into this buffer. We have chosen to use an externally owned buffer, since the received data will not be processed inside the class:

```
class CSocketReader : public CActive
  {
public:
  CSocketReader(CSocketBase& aSocketBase, CBufBase& aLongBuffer)
    : CActive(EPriorityStandard),
      iSocket(aSocketBase.Socket()), iSocketBase(aSocketBase),
      iLongBuffer(aLongBuffer) {}

  void ConstructL();
  ~CSocketReader();

  void StartReading();

protected: // from CActive
  void RunL();
```

```
  void RunError();
  void DoCancel();

private:
  RSocket&            iSocket;
  CSocketBase& iSocketBase;
  CBufBase&    iLongBuffer;
  TBuf8<16384> iShortBuffer; // 16 KByte
  TSockXfrLength      iLastRecvLength;
  };
```

We begin defining the methods with the second-phase constructor and the destructor. The constructor only needs to add the active object to the scheduler. The destructor cancels the active object:

```
void CSocketReader::ConstructL()
  {
  CActiveScheduler::Add(this);
  }

CSocketReader::~CSocketReader()
  {
  Cancel();
  }
```

Reading is started by the public method `StartReading()`. This checks the active object's status and, if it is not active, then issues a new request by calling the `RecvOneOrMore()` method of the socket. This method requires a buffer where the received data is stored and a `TSockXfrLength` reference in which the length of the received data is written after the completion of the read request. The `RecvOneOrMore()` request is completed when any data is received. However, it is not specified whether the sent data is received in one larger burst or in more, smaller ones:

```
void CSocketReader::StartReading()
  {
  if (!IsActive())
    {
    iSocket.RecvOneOrMore(iShortBuffer, 0, iStatus, iLastRecvLength);
    SetActive();
    }
  }
```

Cancelling the receive request is performed by calling `CancelRecv()`:

```
void CSocketReader::DoCancel()
  {
  iSocket.CancelRecv();
  }
```

In the event handler method, if the receive request has been successful, the received data is appended to the buffer that was given at the construction of the socket reader. This buffer is owned by the socket base class, which is notified by calling its `OnReceiveL()` virtual method. Classes derived from `CSocketBase` can process data by accessing the long buffer. Failure of the receive request is handled by the `HandleWriteL()` virtual method of the socket base class:

```
void CSocketReader::RunL()
  {
  switch (iStatus.Int())
    {
    case KErrNone:
      iLongBuffer.InsertL(iLongBuffer.Size(), iShortBuffer);
      iShortBuffer.SetLength(0); // Reset the short buffer
```

```
      StartReading(); // Continue reading

      iSocketBase.OnReceiveL(); // Notify the owner socket base object
    break;

    default:
      iSocketBase.HandleReadErrorL();
    break;
    }
  }
```

7.4.6 The Socket Base Class

The socket base is an abstract class that encapsulates a socket. It allows reading and writing of the socket by using a `CSocketWriter` and a `CSocketReader` instance. Its virtual methods, which were also discussed briefly with the code of the socket reader and writer, notify the derived class when incoming data is received or an error occurs. In SymTorrent, this is the base class of the peer connection class. Generally, it can be used wherever a connected socket is needed. We derive the class from `CActive` so that it can be used to encapsulate other asynchronous requests besides reading or writing the socket. One important use case for this is establishing a connection to another host. Since connecting is also an asynchronous request, the socket base can be used as its active object:

```
class CSocketBase : public CActive
  {
public:
  CSocketBase() : CActive(EPriorityStandard) {}
  void ConstructL();
  void ConstructL(RSocket& aSocket);
  ~CSocketBase();

public:
  void SendL(const TDesC8& aDes);
  inline RSocket& Socket() { return iSocket; }

protected:
  virtual void OnReceiveL() = 0; // Called when incoming data is received
  virtual void HandleReadErrorL() = 0; // Called on read error
  virtual void HandleWriteErrorL() = 0; // Called on write error

protected:
  RSocket iSocket;
  CBufFlat* iRecvBuffer; // Buffer passed to the socket reader
  TBool iIncomingConnection;

private:
  CSocketReader* iSocketReader;
  CSocketWriter* iSocketWriter;

  friend class CSocketReader; // for calling HandleReadErrorL()
  friend class CSocketWriter; // for calling HandleWriteErrorL()
  };
```

The class has two second-phase constructors. The one without parameters is used in the general case when we want to establish the connection to another host. By contrast, the other overload takes an already connected socket handle. This can be used when an incoming connection is accepted (i.e. by using our previously written socket listener). The already connected socket is attached to the socket base instance. There is a protected member variable, `iIncomingConnection`, that enables us to check whether this socket was initialized with an incoming connection. In the constructor, the network connection is started, the socket is initialized, and then the socket reader and writer instances are created. Reading from the socket is started immediately:

```
void CSocketBase::ConstructL()
  {
```

```
  if (!NetMgr()-> IsNetworkConnectionStarted())
    NetMgr()->StartNetworkConnectionL();

  if (!iIncomingConnection)
    iSocket.Open(NetMgr()->SocketServ(), KAfInet,
      KSockStream, KProtocolInetTcp, NetMgr()->NetworkConnection());

  iRecvBuffer = CBufFlat::NewL(256); // 256 byte granularity of buffer expansion

  iSocketReader = new (ELeave) CSocketReader(*this, *iRecvBuffer);
  iSocketReader->ConstructL();
  iSocketReader->StartReading();

  iSocketWriter = new (ELeave) CSocketWriter(*this);
  iSocketWriter->ConstructL();
  }

void CSocketBase::ConstructL(RSocket& aSocket)
  {
  iIncomingConnection = ETrue;
  iSocket = aSocket;

  ConstructL(); // Call the other overload
  }
```

To send data, the socket writer instance can be used. To make the operation simpler, we can create a `Send()` method that calls the `WriteL()` method of the socket writer. It should be noted that, since this is an asynchronous operation, the program is not stopped at this point. The `Write()` request is forwarded to the socket server, and sending of data takes place asynchronously in another process. In this implementation, the socket base class is not notified when a write request is completed; however, it is not a difficult task to add such a function to `CSocketWriter`:

```
void CSocketBase::SendL(const TDesC8& aDes)
  {
  iSocketWriter->WriteL(aDes);
  }
```

The socket base can be used for several purposes. The derived classes must implement its three pure virtual methods, most importantly the `OnReceiveL()` method. The example here does not establish a connection. Connecting can be implemented by using the `Connect()` method of the socket handle (`RSocket`).

7.4.7 The Peer Connection

Now that we have a framework that enables us to start network connections, listen for incoming TCP/IP connections, and send/read data on the sockets, we can move on to creating the class that encapsulates a peer connection. In BitTorrent, the content of the torrent is downloaded from the peers. We do not have the space to go into the peer communication protocol specification or the complex logic of the class. Instead, we will show some of the more interesting methods of the class as examples. The peer connections (`CSTPeerConnection`) are associated with a torrent. On the code level, the `CSTPeerConnection` instances are owned by a `CSTTorrent` instance. The complete version of these classes can be found in the project SymTorrentEngine. All the names of the classes in the engine are prefixed with ST, which refers to SymTorrent.

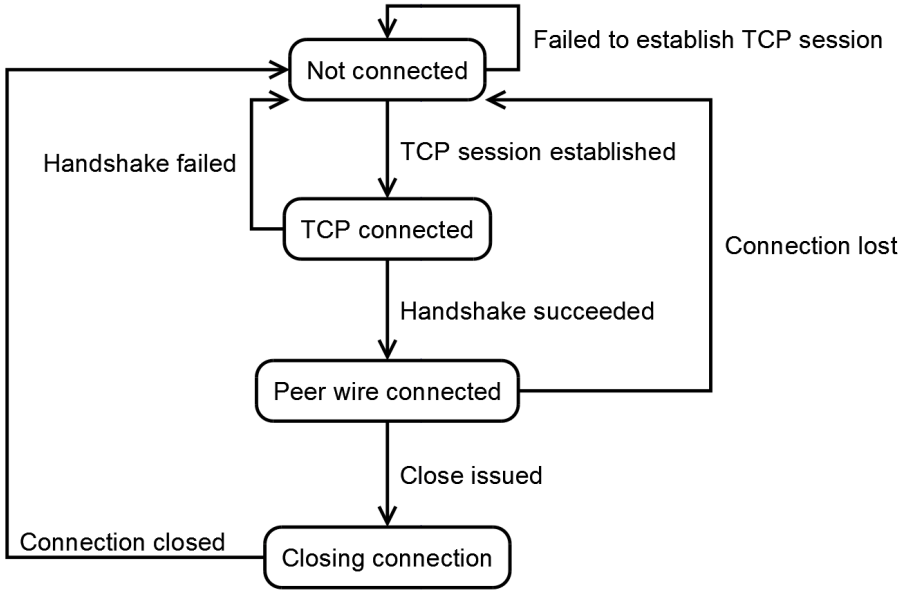

Figure 7.8 State chart of a peer connection

A state chart of a peer connection life cycle can be seen in Figure 7.8. Newly created peer connections are not connected. The current state is stored in the `iState` member variable. This is an enumeration, referred to as `TPeerConnectionState`. The various values it can take are as follows:

- `EPeerNotConnected`: the peer is not connected.
- `EPeerTcpConnecting`: the peer is establishing a TCP connection.
- `EPeerPwHandshaking`: the peer is performing the BitTorrent (peer-wire) handshake.
- `EPeerPwConnected`: the peer is connected (the handshake is complete).
- `EPeerClosing`: the peer is disconnected and freeing up its resources.

As you can see, a newly created, unconnected peer connection enters the listed states in a sequential order. Firstly, it establishes a connection, and then it performs the handshake. If the handshake has been successful, it enters the connected state and exchanges messages until the connection is closed. The protocol of the handshake and the messages are fixed, and its specification can be found at the official BitTorrent site [2]. Without going into such detail, we will investigate the `OnReceiveL()` method of the class, which is called when incoming data is received. If the peer is still in the handshake state, then the size of the received data is checked. If the whole handshake string is received, then parsing the handshake can be performed. You can see that the received data is read from `iRecvBuffer`, the buffer that we also passed to the socket reader. The length of the buffer can be queried, along with the data contained.

Receiving messages in the connected state works similarly. The first step is to check the length of the next message, which is transmitted as a four-byte integer. All of the BitTorrent messages begin with this length prefix. Having at least as many bytes in the receive buffer as the parsed message length means that a new message has been received, and can be parsed. It is very important that, after processing the message, it be deleted from the receive buffer so that the next message can be processed:

```
void CSTPeerConnection::OnReceiveL()
  {
  switch (iState)
    {
```

```
    case EPeerPwHandshaking:
      {
      TInt protLength = (iRecvBuffer->Ptr())[0];
      TInt handshakeLength = protLength + 1 + 48;
      if (iRecvBuffer->Size() >= handshakeLength)
        {
        // Parse handshake...
        }
      }
    break;

    case EPeerPwConnected:
      {
      while (iRecvBuffer->Size() >= 4)
        {
        TUint messageLength = ReadInt();
        if (TUint(iRecvBuffer->Size()) >= (4 + messageLength))
          {
          // Process incoming message...

          // Delete the processed data from the buffer
          iRecvBuffer->Delete(0, 4 + messageLength);
          }
        else
          break;
      }
    break;
    }
  }
```

We used the method `ReadInt()` in the implementation of `OnReceiveL()`. This method shows how a four-byte-long integer can be extracted from the buffer:

```
TUint CSTPeerConnection::ReadInt(TInt aIndex)
  {
  TPtrC8 ptr = iRecvBuffer->Ptr().Right(
    iRecvBuffer->Ptr().Length() - aIndex);

  TUint value = ptr[0] << 24;
  value += (ptr[1] << 16);
  value += (ptr[2] << 8);
  value += ptr[3];

  return value;
  }
```

Sending an integer can be performed by separately sending all of its four bytes:

```
void CSTPeerConnection::SendIntL(TUint32 aInteger)
  {
  TBuf8<4> buffer;
  buffer.SetLength(4);

  buffer[3] = aInteger & 0xFF;
  buffer[2] = ((aInteger & (0xFF << 8)) >> 8);
  buffer[1] = ((aInteger & (0xFF << 16)) >> 16);
  buffer[0] = ((aInteger & (0xFF << 24)) >> 24);

  SendL(buffer);
  }
```

Closing and deleting a peer connection is performed in two steps. As mentioned earlier, the peer connection instances are created and owned by a `CSTTorrent` instance. Thus, it is also the owner's responsibility to delete the instances. Also, there are various tasks that have to be performed when a peer is disconnected, such as trying to establish new connections or removing the peer's address if it has failed too many times. To do so, peers close their connection and set the value of the variable that determines the tasks that have to be

performed. This variable, which is an instance of the enumeration `TConnectionCloseOrder` and is referred to as `iCloseOrder`, can be set to the following values:

- `EDeletePeer`: the peer has to be deleted.
- `EIncreaseErrorCounter`: the error counter of the peer has to be increased.
- `EDelayReconnect`: the peer should be reconnected after a short delay.
- `ENotSpecified`: no specified order.

After setting `iCloseOrder`, the state of the peer is changed to closing. After this, the peer is not deleted immediately. Instead, it will be deleted by the `CSTTorrent` owner when it becomes aware that there is a peer that is closing. `CSTTorrent` periodically checks its peers to see whether they are closing:

```
void CSTPeerConnection::CloseL(TConnectionCloseOrder aOrder)
  {
  if (iState != EPeerClosing)
    {
    iCloseOrder = aOrder;

    iSocketReader->Cancel();
    iSocketWriter->Cancel();
    iSocket.Close();

    ChangeState(EPeerClosing);
    }
  }
```

7.4.8 The Tracker Connection

The tracker is a fundamental component of the BitTorrent system. It is responsible for coordinating the whole swarm and supplying the peer addresses to its clients. Each BitTorrent client must periodically establish connections to the tracker to obtain new peer addresses and announce its presence to the swarm. The clients communicate with the tracker via standard HTTP GET requests. Fortunately, Symbian OS offers a framework that makes issuing HTTP requests an easy procedure. Here, we show the class `CSTTrackerConnection`, which is responsible for announcing to the tracker in SymTorrent. To be able to receive HTTP events, the `MHTTPTransactionCallback` interface must be implemented. Its methods are called by the framework when an event is received during the HTTP request. This class is also part of the project SymTorrentEngine:

```
class CSTTrackerConnection : public CBase,
                             public MHTTPTransactionCallback
  {
public:
  enum TDownloadResult
    {
    EPending = 0,
    EFailed,
    ESucceeded
    };

  CSTTrackerConnection(CSTTorrent& aTorrent,
    TTrackerConnectionEvent aEvent = ETrackerEventNotSpecified)
  : iTorrent(aTorrent), iEvent(aEvent) {}

  void ConstructL();
  ~CSTTrackerConnection();
  void StartTransactionL();
  void Cancel();
  TBool IsRunning() const;
  TDownloadResult Result() { return iResult; }
  TTrackerConnectionEvent Event() const;

private:
```

```
  void SetHeaderL(RHTTPHeaders aHeaders, TInt aHdrField,
    const TDesC8& aHdrValue);
  void SetFailed();
  void CreateUriL();

private: // from MHTTPSessionEventCallback
  void MHFRunL(RHTTPTransaction aTransaction, const THTTPEvent& aEvent);
  TInt MHFRunError(   TInt aError,   RHTTPTransaction aTransaction,
    const THTTPEvent& aEvent);

private:
  CSTTorrent& iTorrent;
  RHTTPSession iSession;
  RHTTPTransaction iTransaction;
  TDownloadResult iResult;
  HBufC8* iUri;
  HBufC8* iReceiveBuffer;
  TTrackerConnectionEvent    iEvent;
  };
```

The various states through which the system goes while performing the HTTP transaction are depicted in Figure 7.9.

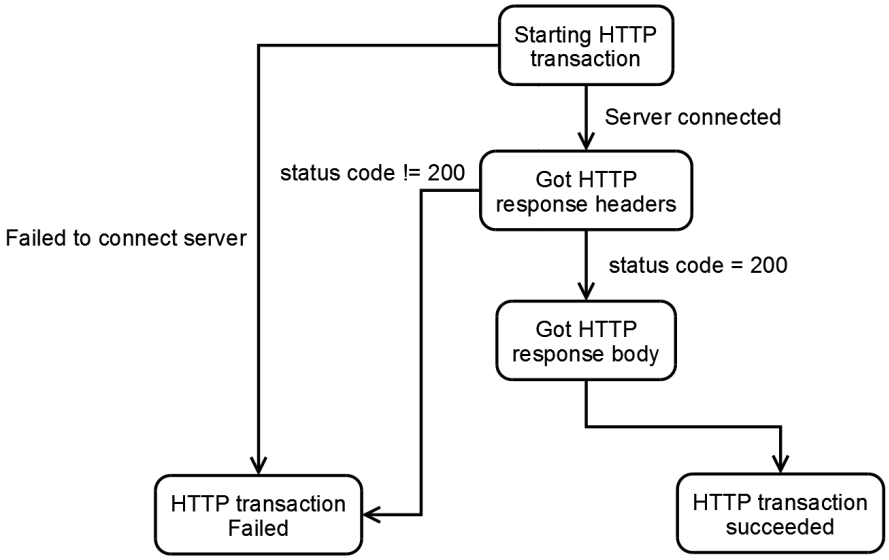

Figure 7.9 HTTP transaction state chart

Starting the actual HTTP transaction is carried out by the method `StartTransactionL()`. The first part of the method might seem to be a little complicated. It is responsible for setting the various parameters of the HTTP session, including the preferred network connection. The result is the same as with the sockets: no network connection selection dialog is popped up, and it uses the network connection we set up before. After setting these basic properties, listening on a socket is started. This step is not required to establish a connection to the tracker, but we should be aware that other peers will try to connect us after we have connected to the tracker. Thus, this is the right point to start listening. The last step is to create the URI (a resource identification string) of the request and configure some basic headers, such as the name of the user agent and the type of acceptable content. Finally, the request is issued by calling `SubmitL()` on the HTTP transaction:

```
void CSTTrackerConnection::StartTransactionL()
  {
  iSession.OpenL(); // Open HTTP session
  RStringPool strP = iSession.StringPool();
  RHTTPConnectionInfo connInfo = iSession.ConnectionInfo();
```

```
  // Set the socket server property
  connInfo.SetPropertyL(strP.StringF(HTTP::EHttpSocketServ,
    RHTTPSession::GetTable()), THTTPHdrVal(NetMgr()->SocketServ().Handle()) );

  // Set the network connection property
  TInt connPtr = REINTERPRET_CAST(TInt, &(NetMgr()->NetworkConnection()));
  connInfo.SetPropertyL(strP.StringF(HTTP::EHttpSocketConnection,
    RHTTPSession::GetTable()), THTTPHdrVal(connPtr));

  // Start listening before connecting to the tracker
  NetMgr()->StartListeningL(0, Preferences()->IncomingPort());

  // Create URI string for the HTTP request
  CreateUriL();

  TUriParser8 uri;
  uri.Parse(*iUri);
  RStringF method = iSession.StringPool().StringF(HTTP::EGET,
    RHTTPSession::GetTable());
  iTransaction = iSession.OpenTransactionL(uri, *this, method);

  RHTTPHeaders hdr = iTransaction.Request().GetHeaderCollection();
  _LIT8(KUserAgent, "SymTorrent");
  SetHeaderL(hdr, HTTP::EUserAgent, KUserAgent);
  _LIT8(KAccept, "*/*"); // Accept all content type
  SetHeaderL(hdr, HTTP::EAccept, KAccept);

  // Submit the transaction.
  iTransaction.SubmitL();
  iRunning = ETrue;
  }
```

After starting the HTTP GET request, events will be received according to the various states of the transaction. These can be handled in the virtual method `MHFRunL()`. There are several events, the most important of which are as follows:

- `THTTPEvent::EGotResponseHeaders`: the HTTP headers are received. Here we should check the status code of the transaction, which reflects whether the request has been successful. In the example, we check whether the status code 200 is received. We can also check the various headers that are received in the response.
- `THTTPEvent::EGotResponseBodyData`: this event is triggered when a part of the body of the response is received. The body can be accessed via the `Body()` method of the transaction. In this example, we create a data buffer and append the received body part to it. After we have processed the received data, it must be released by calling `ReleaseData()` on the body. This event could be triggered multiple times, based on how many parts of the response are received.
- `THTTPEvent::ESucceeded`: this event means that the HTTP transaction has succeeded. Here we can process the received data, in our case the list of peers sent by the tracker. A method of `CSTTorrent` is called that processes the response and registers the received peer addresses.
- `THTTPEvent::EFailed`: this event is triggered if the transaction fails. In SymTorrent, this results in closing the transaction and setting the state of the class to failed.

```
void CSTTrackerConnection::MHFRunL(RHTTPTransaction aTransaction,
  const THTTPEvent& aEvent)
  {
  switch (aEvent.iStatus)
    {
    case THTTPEvent::EGotResponseHeaders:
      RHTTPResponse resp = aTransaction.Response();
      TInt status = resp.StatusCode();

      if (status != 200)
        {
```

```
        SetFailed();
        Cancel();
        }
    break;

    case THTTPEvent::EGotResponseBodyData:
      MHTTPDataSupplier* body = aTransaction.Response().Body();
      TPtrC8 dataChunk;
      body->GetNextDataPart(dataChunk);

      if (iReceiveBuffer)
        {
        HBufC8* temp = HBufC8::NewL(iReceiveBuffer->Length() + dataChunk.Length());
        TPtr8 tempPtr(temp->Des());
        tempPtr.Copy(*iReceiveBuffer);
        tempPtr.Append(dataChunk);

        delete iReceiveBuffer;
        iReceiveBuffer = temp;
        }
      else
        iReceiveBuffer = dataChunk.AllocL();

      body->ReleaseData();
    break;

    case THTTPEvent::ESucceeded:
      iResult = ESucceeded;

      CSTBencode* bencodedResponse = CSTBencode::ParseL(*iReceiveBuffer);

      if (bencodedResponse)
        {
        CleanupStack::PushL(bencodedResponse);
        iTorrent.ProcessTrackerResponseL(bencodedResponse);
        CleanupStack::PopAndDestroy(bencodedResponse);
        }

      aTransaction.Close();
      iRunning = EFalse;
    break;

    case THTTPEvent::EFailed:
      SetFailed();
      aTransaction.Close();
      iRunning = EFalse;
    break;

    default:
      if (aEvent.iStatus < 0)
        {
        SetFailed();
        aTransaction.Close();
        iRunning = EFalse;
        }
    break;
    }
  }
```

7.4.9 The Torrent

The torrent class (`CSTTorrent`) is one of the more complex classes of SymTorrent, and thus it cannot be discussed in full detail here. The torrent establishes connections to the peers, and it is also responsible for periodically issuing tracker announces via a `CSTTrackerConnection` instance. The main part of the application logic is implemented in the `OnTimerL()` method which is triggered by a timer every second. The seconds passed since opening the torrent is counted by `iEllapsedTime`. The first part is responsible for maintaining the tracker connection: if there is an active connection, then its result is checked. If the tracker connection fails, then a new request is issued until the retry limit is reached (in our case, 10). In the second part of the method, two main tasks are carried out. Firstly, the tracker is

announced if the specified timeout limit is reached. Secondly, the peers are updated and new peer connections are established if needed. `CSTPeer` is a class that stores the address and some general properties of a peer. When the peer is connected, a `CSTPeerConnection` instance is created:

```
void CSTTorrent::OnTimerL()
  {
  iEllapsedTime++;

  if (iTrackerConnection)
    {
    iTrackerConnection->OnTimerL();

    switch (iTrackerConnection->Result())
      {
      case CSTTrackerConnection::ESucceeded:
        iTrackerFailures = 0;

        iLastTrackerConnectionTime.HomeTime();
        delete iTrackerConnection;
        iTrackerConnection = NULL;
      break;

      case CSTTrackerConnection::EFailed:
        iTrackerFailures++;

        delete iTrackerConnection;
        iTrackerConnection = NULL;

        if (iActive && (iTrackerFailures < 10))
          AnnounceL();
       break;
      }
    }

  if (iActive)
    {
    if ((iEllapsedTime % iTrackerRequestInterval) == 0)
      AnnounceL();

    for (TInt i=0; i<iPeers.Count(); i++)
      {
      CSTPeer* peer = iPeers[i];
      peer->OnTimerL();

       if (peer->State() != EPeerNotConnected)  // The peer is connected
         activeConnectionCount++;
       else  // The peer is not connected
         if (!iComplete) && (activeLocalConnectionCount < KMaxPeerConnectionCount))
           {
           peer->ConnectL(*this, iTorrentMgr);
           activeConnectionCount++;
           }
      }
    }
  }
```

7.4.10 The Torrent Manager

The torrent manager (`CSTTorrentManager`) is the central singleton class of SymTorrent's engine. It is responsible for creating the torrents and other system-level tasks. Here, we show the method that loads a new torrent file and adds it to the engine. Torrent files are loaded by the `LoadL()` method of `CSTTorrent`. If loading the file has been successful, then the newly created `CSTTorrent` instance is added to the array of torrents, and it immediately starts downloading:

```
TInt CSTTorrentManager::OpenTorrentL(const TDesC& aFileName)
  {
  CSTTorrent* torrent = new(ELeave) CSTTorrent(this);
```

```
CleanupStack::PushL(torrent);
torrent->ConstructL();
TInt loadResult = torrent->LoadL(aFileName);

if (loadResult == KErrNone)
  {
  iTorrents.AppendL(torrent);
  CleanupStack::Pop(); // torrent

  torrent->Start(); // Start downloading the torrent
  }
else
  {
  CleanupStack::PopAndDestroy(); // torrent
  return loadResult;
  }
}
```

7.4.11 Differences in GridTorrent

As stated before, GridTorrent is built on top of the same engine as SymTorrent. Actually, we extended SymTorrent's engine with the features needed by GridTorrent. The most notable differences are in network connection handling and in the application logic of the peer connections. On the network connection level, SymTorrent requires only one type of active connection (e.g. 3G). In contrast to this, GridTorrent communicates with the local peers over a different network connection to that used for the standard peers acquired from the tracker. This means that the network manager needs to be able to handle several network connections, and these must be made available for the different networking objects, such as sockets. Another problem was that we had to add support for Bluetooth connections, which are handled somewhat differently to WLAN/3G/GPRS. Although Bluetooth also uses sockets, it is not supported by the access point framework and must be initialized in a completely different way. In GridTorrent, Bluetooth connections are used just like the other connections; most of the differences are handled in the network manager. However, there is one key difference: the nature of how Bluetooth networks work. Currently, devices based on Symbian OS support only the Bluetooth piconet scheme. Piconets have one master peer and up to seven slave peers. Connections can only be established by the master. GridTorrent supports both this point-to-multipoint scheme and the standard IP-based WLAN/GPRS networks. If the local connection is Bluetooth, the peer must choose between slave and master modes. If WLAN is used, connection can be established between the peers in both directions.

In addition to the network connection layer, peer connections and the piece selection strategy also work differently in GridTorrent. We introduced a couple of new messages that enable the peers in the local cluster to inform each other about their progress. Since peers know which pieces are available in the local cluster, they can focus on downloading the rarer pieces from the Internet. The piece selection strategy needs further work, but current results show that even this simple algorithm considerably increases the performance of the swarm.

7.5 Conclusion

In this chapter we have shown how a complex peer-to-peer application handles network connections and sockets. We have discussed the basics of the BitTorrent protocol and analyzed a small part of the source code of a client written in Symbian C++. We have also outlined the concepts behind GridTorrent, the world's first BitTorrent client that utilizes local cooperation to save energy and increase transfer speed. Although we have not been able to discuss every part in detail, the source code of SymTorrent is freely available for anyone who is looking for a deeper insight into programming mobile peer-to-peer clients.

References

[1] Cohen, B., '*Incentives Build Robustness in BitTorrent*'. In Proceedings of the 1st Workshop on Economics of Peer-to-Peer Systems, Berkeley, CA, June 2003.

[2] Cohen, B., '*The BitTorrent Protocol Specification*'. Available at: http://www.bittorrent.org/beps/bep_0003.html [accessed 6 September 2008].

[3] Loewenstern, A., '*DHT Protocol*'. Available at: http://www.bittorrent.org/beps/bep_0005.html [accessed 6 September 2008].

[4] Kelényi, I., '*SymTorrent webpage*'. Available at: http://symtorrent.aut.bme.hu [accessed 25 February 2009].

[5] Willee, H., '*Symbian OS Support for Writeable Static Data in DLLs v2.3*'. The Symbian Developer Network. Available at: http://developer.symbian.com/main/downloads/papers/static_data/SupportForWriteableStaticDataInDLLs.pdf [accessed 25 February 2009].

[6] Morris, B., '*CActive and Friends v1.89*'. The Symbian Developer Network. Available at: http://developer.symbian.com/main/downloads/papers/CActiveAndFriends/CActiveAndFriends.pdf [accessed 25 February 2009].