



Contents

- 1 Introduction
 - ◆ 1.1 Using client-server
 - ◆ 1.2 What we could put into thread function
- 2 Indivisible synchronous operation
 - ◆ 2.1 Few notes about thread use
 - ◆ 2.2 Architecture
 - ◆ 2.3 Implementation
- 3 Periodical synchronous operation with Rendezvous() call
 - ◆ 3.1 Architecture
 - ◆ 3.2 Implementation
 - ◆ 3.3 Enhancing previous solution with message queue RMsgQueue
- 4 Example source code

Introduction

In this tutorial I will show how to process synchronous operation in second thread and two different approaches to inform the main UI thread about the operation status (message queue and thread rendezvous.).

Mobile phones must execute as many operations as possible asynchronously to give the user the real time experience, by which I mean that application UI must react on user input immediately. This also predicates that it is not possible to hang-up the GUI until synchronous long-term operation is finished. Such behavior is not only acceptable by users, but it could raise another unacceptable behavior like **ViewSvr 11** panic, which occurs when ViewSvr active objects do not respond in the time up to 10-15 seconds. Anyway, sometimes application has to execute long running synchronous operations, like searching in database, processing XML, files, using code ported from other environments. It also happens that a solution was developed, but it was tested on small amount of data, so when application comes to the "real live" it becomes clear that it is not ideal from the execution time perspective. Then there are two possible ways how to solve it: It can be reworked or some "dirty solution" could be applied to not cut down the execution time, but to make it asynchronous.

The nature of the Symbian is using the client-server architecture and active objects. This allows applications in most cases to be single threaded, because operations are executed by another processes, which inform applications once the operation is completed. In spite of this sometimes we simply need to execute some operations directly in the application ? in such case we normally display a wait (**CAknWaitDialog**) or progress dialog (**CAknProgressDialog**). The difference between them is that progress dialog is showing how big portion of the operation is complete, while the wait dialog is just an information, that something is executed in the background and user has to wait. In typical case we recognize two different synchronous operations:

- long running task, which cannot say how many of the operation is complete
- long running task, which is divided into many small operations and it is known how many times it will be executed (typically for or while cycle)

I will show in this article how to make those operations to be executed asynchronously, while the dialog will be displayed in the UI.

Using client-server

It is quite common that the UI application is accompanied by the EXE server. Such servers are sometimes launched after phone boot, but more often they are launched when the application is for the first time started by client. The normal behavior is that the method **Connect()** of the **RSessionBase** derived object opens the connection to the server and if the server is not running it starts it. Such connections are usually started when the application is being initialized and unwanted outcome of such approach is that if the server is not already running its startup could take few seconds, during which the application UI do not redraw and application have a long startup time. We can easily solve this by connecting to the server (if it is not already running) in the second thread:

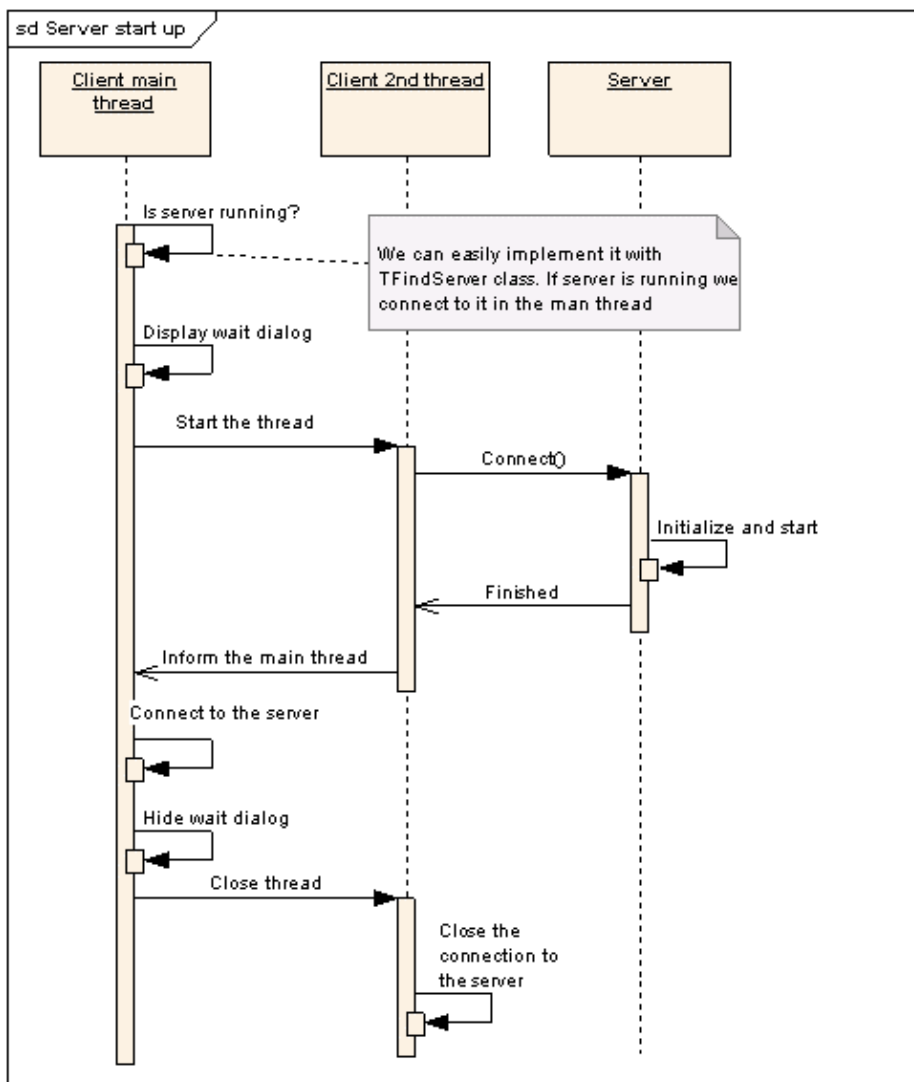


Figure 1. Server start-up in the second thread

What we could put into thread function

Sometimes it is not so clear if we should execute in the new thread the synchronous operation or if would be better to left it in the main thread, while the second thread will display the dialog. Generally the preferable way is to left all the GUI stuff in the main thread, which initializes the whole framework (**CEikonEnv**, **CAknEnv**) and which communicates with the window server (**WSERV**), view server, etc. because it cares of all GUI and key handling, which will be impossible to make on your own in the new thread (additionally not all client-server session could be shared across threads).



Figure 2. Wait and progress dialog

Indivisible synchronous operation

This section will describe the procession of indivisible synchronous operation (I will use **User::After()** method to simulate such operation), while the wait dialog will be displayed in the foreground. If anything will be unclear, use the attached example. (Note that it has assigned the UID from unprotected range and it does not use any capability, so it can be self signed, issuing developer certificate is not necessary).

Few notes about thread use

Using the thread for executing asynchronous operation you have to be aware of following issues:

- All handles to **RThread** must be closed properly.
- Once the **RThread::Close()** is called the handle becomes NULL, calling any method on not initialized or closed **RThread** object will cause **KERN EXEC 0** panic, so you have to do that very carefully.
- The method, which starts the asynchronous operation within thread should return the error value if started twice. Method **RThread::Create()** will return **KErrAlreadyExists** value, but for checking if the thread is actually being executed you cannot use the **RThread** object, which is already assigned to the running thread, because the **Create()** method will change it.
- If the new thread allocates memory and both the main thread and the second thread share the same heap (**RHeap**) and the indivisible synchronous operation will be cancelled (i.e. the thread will be killed) it will cause a memory leak.

- When the class **TFindThread** is used for searching for thread object, the wildcard must be used, because the thread full name consists of the application name, UID3 and the thread name (e.g. "TL wiki[043a5375]0001::ThreadFNWiki").

Architecture

The main functionality is wrapped into class **CThreadLoader** and **CThreadLoaderAO**:

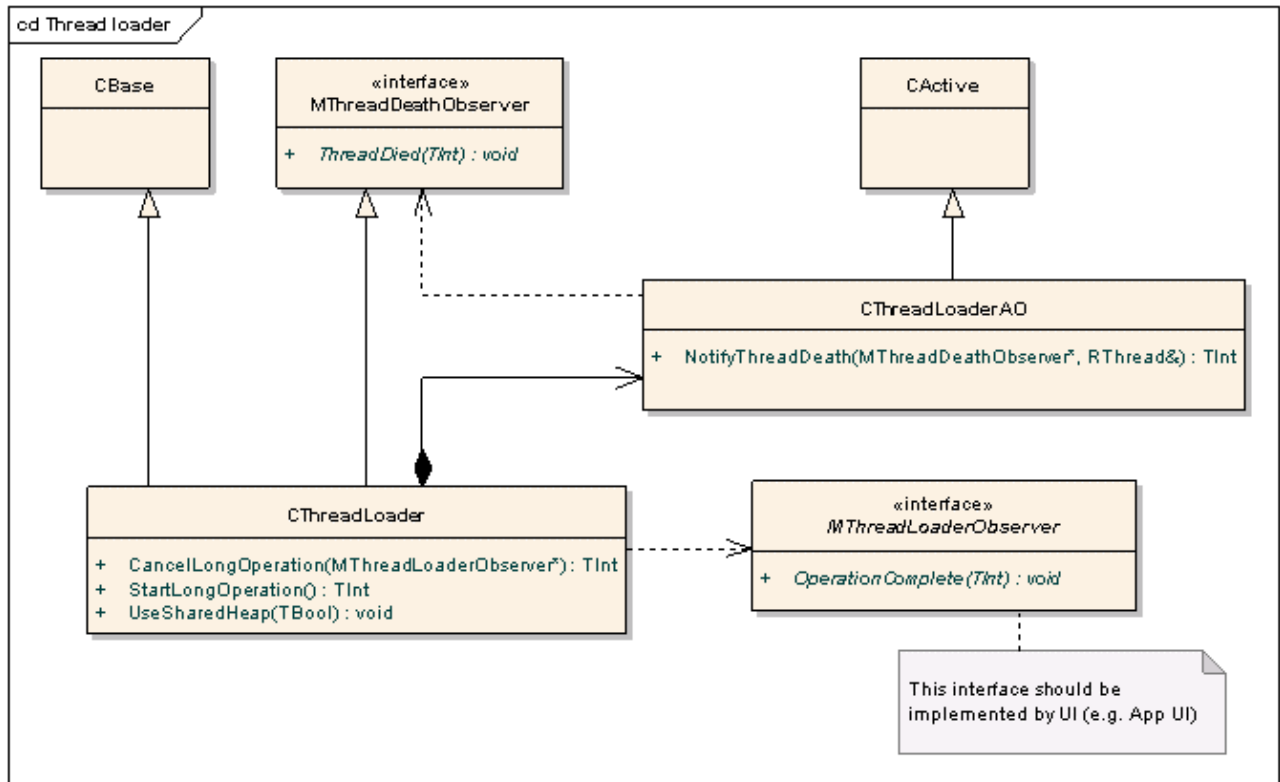


Figure 3. UML schema for synchronous indivisible operation

When the new thread is created, in most cases you have to create the cleanup stack and start the synchronous operation in the method under **TRAPD** macro. In the main thread is in the same time displayed the wait dialog. When the operation is finished the method **OperationComplete()** is called. It could be unclear what exactly means that thread is finished and when it should be called? Where is exactly the point to call this method? On the first sight you could say that it can be called after the synchronous operation is finished, i.e. before method **LongOperationThread()** reach its end. Unfortunately such straightforward solution has following drawbacks:

- **LongOperationThread()** method body is still executed within the new thread, which limits its use, e.g. we cannot hide the wait dialog in callback **OperationComplete()**...
- we must close thread handle
- in the case the thread will be killed **OperationComplete()** callback will not be called

The solution for problems mentioned above is use **RThread::Logon()** thread request.

Few notes about thread use

Implementation

Start thread: We create the thread, invoke **RThread::Logon()** asynchronous request and resume the thread. Consequently OS starts **LongOperationThread()** within the new thread. There is created the cleanup stack and under **TRAPD** macro is invoked the synchronous operation. We do not use Active Scheduler in our examples.

```
TInt StartLongOperation( MThreadLoaderObserver* aThreadLoaderObserver )
{
    // Save the observer pointer
    iThreadLoaderObserver = aThreadLoaderObserver;

    // Create thread
    RThread thread;

    TInt err = KErrNone;
    if ( iUseSharedHeap )
        err = thread.Create( KThreadID, StartLongOperationThread, KDefaultStackSize, &User::Heap );
    else
        err = thread.Create( KThreadID, StartLongOperationThread, KDefaultStackSize, KMinHeapSize );

    if ( err == KErrNone )
    {
        iThread = thread;
        iThread.SetPriority( EPriorityMuchMore );

        // Set the death notifier
        err = iAO->NotifyThreadDeath( iThread, this );

        if ( err == KErrNone )
        {
            // Resume the thread code
            iThread.Resume();
        }
        else
        {
            iThread.Close();
            return err;
        }
    }
    return err;
}

TInt CThreadLoader::StartLongOperationThread( TAny *aPtr )
{
    // Allocates and constructs a cleanup stack
    // Note: This is not necessary in all cases, but mostly you will need it
    CTrapCleanup* cleanup = CTrapCleanup::New();

    if ( cleanup )
    {
        // Execute trapped thread code safely
        TRAPD( err, StartLongOperationThreadTrappedL( NULL ) )

        // The cleanup stack must be deleted
    }
}
```

Synchronous_operations

```
        delete cleanup;

        return err;
    }
    else
        return KErrNoMemory;
}
```

Cancel thread: When the user decides to kill the thread, we try to find it and then we simply terminate it with **KErrCancel** reason. Just after that the **CThreadLoaderAO::RunL()** method is called and UI is informed that thread does not live longer.

```
TInt CancelLongOperation()
{
    TFullName threadName;

    TFindThread findThread( KThreadIDWildcard );
    TInt err = findThread.Next( threadName );

    if ( err == KErrNone)
    {
        // Kill the thread and close the handle
        iThread.Kill( KErrCancel );
        iThread.Close();
    }
    return err;
}
```

Synchronous_operations

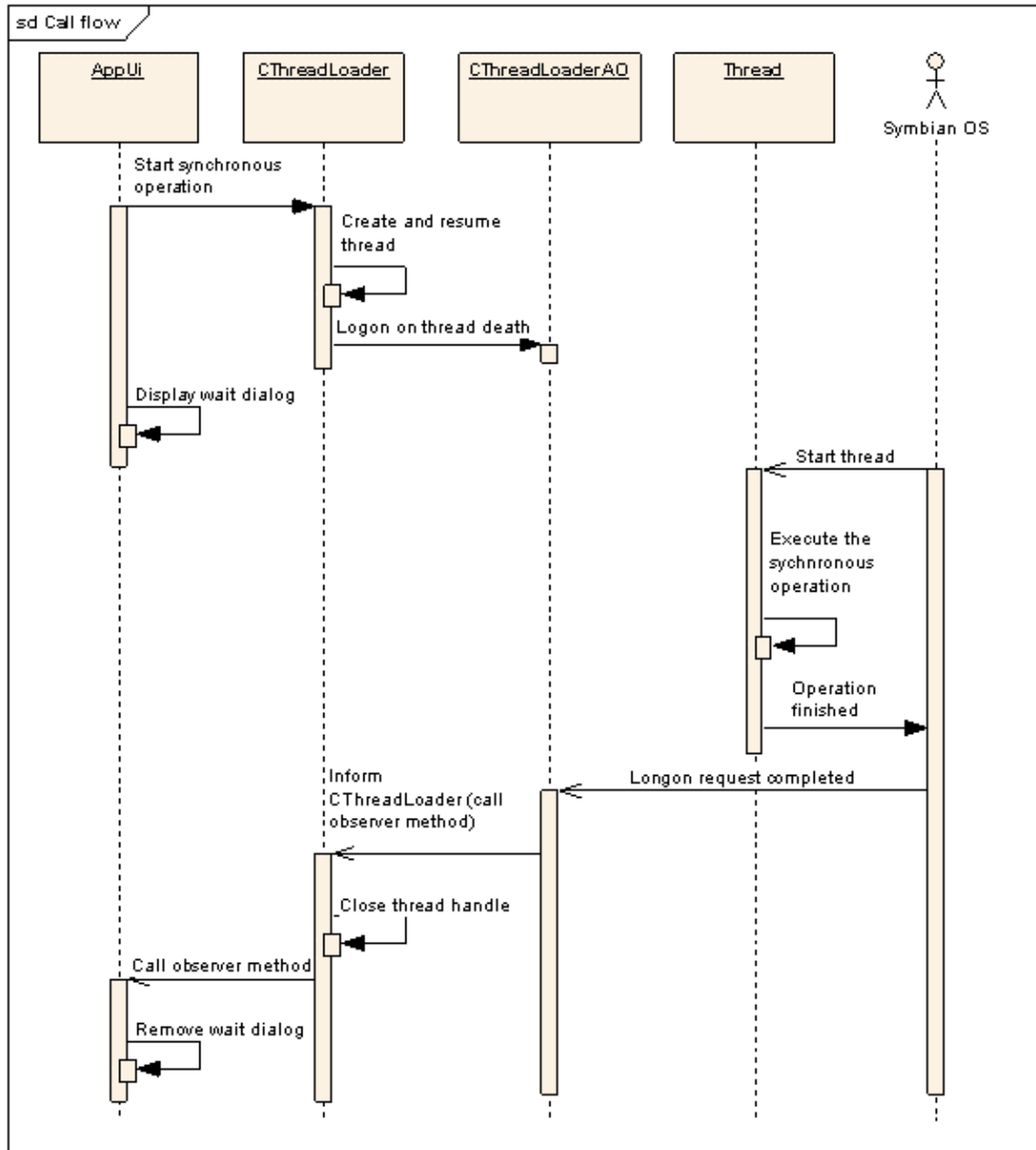


Figure 4. Call flow for synchronous indivisible operation

Note that we never call `RThread::LogonCancel()` method. It is not necessary, because we are guarding the thread termination, so it can never happen that we will delete the active object with an outstanding request.

Periodical synchronous operation with Rendezvous() call

In this paragraph I will show how we could update the progress dialog in the main thread. For this purpose I will use the for cycle with synchronous operation.

Architecture

Firstly we need to extent the **CThreadLoader** class with new methods. In the first attempt we will try to use rendezvous IPC mechanism. It allows communication between two threads via active objects. Unfortunately calling **RThread::Rendezvous()** is not intended to be used for such situation, so its use is little bit tricky, it needs to call **User::After(1)** before the **Rendezvous()** method execution in each for cycle and additionally we must change the priority of the second thread to **EPriorityMuchMore**. This allows the thread to have enough time update the rendezvous value, otherwise **CThreadLoaderRendezvousAO::RunL()** method will be executed in infinite cycle with the initial value. Another drawback of this attempt is that not every **Rendezvous()** call is executed, because the AO could process the incoming value slower then the working thread loop. This becomes obvious, when during the updating of the **CAknProgressDialog** we use the method **CEikProgressInfo::IncrementAndDraw()** with parameter set to value 1 (one) instead of the method **CEikProgressInfo::SetAndDraw()** with the parameter equal to the current for loop index. This problem could be solved by using **RMutex**, but this could hang-up the main UI thread, so I recommend to not use it. Another "dirty solution" could be, that the working thread is updating some value which is in the timer read in the main thread.

When we want to cancel the thread we just set the **TBool** flag instead of killing the thread like in pervious case, the for loop is consequently leaved and the thread execution is stopped.

Changes in UML are marked by red color:

Synchronous_operations

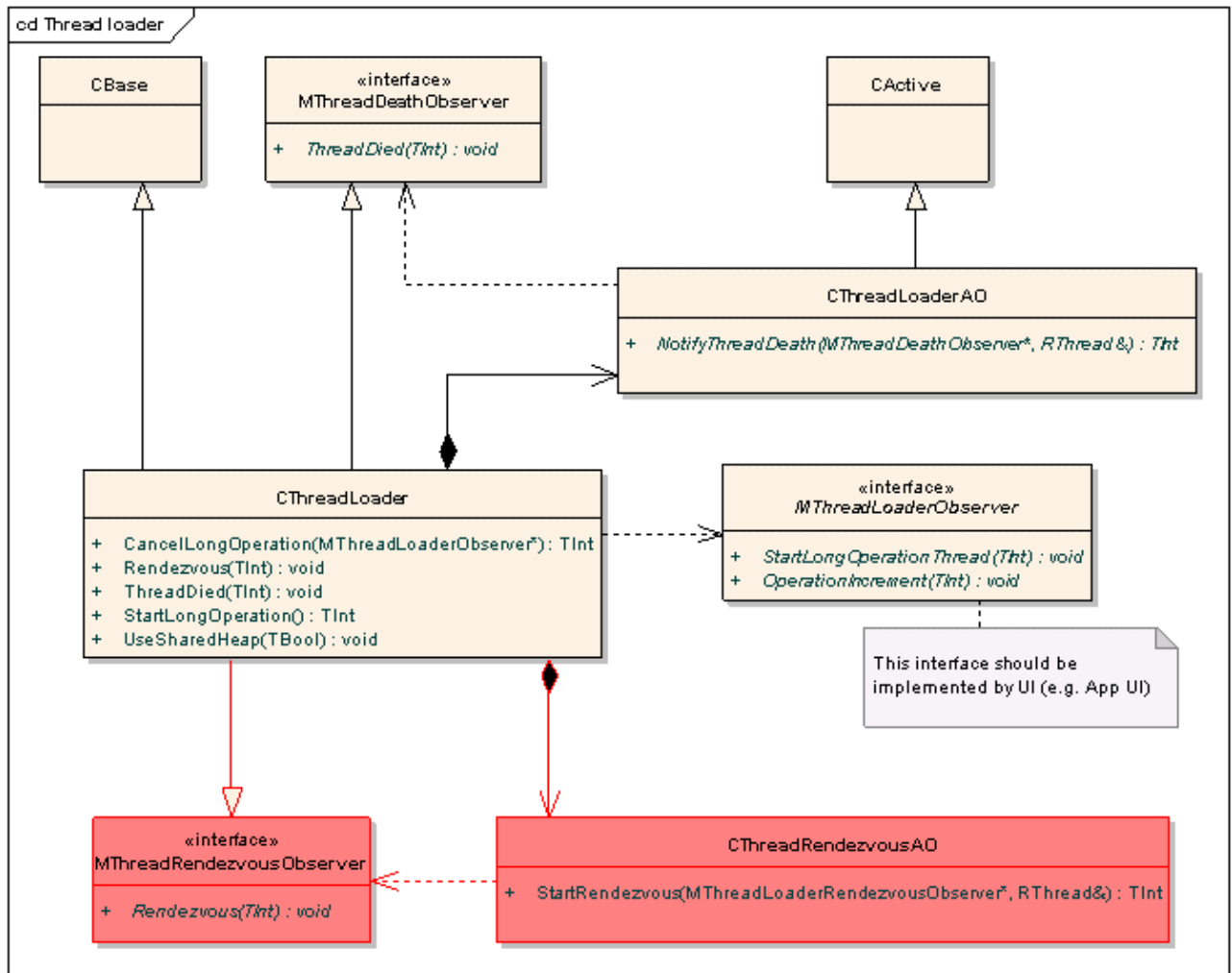


Figure 5. UML schema for periodical synchronous operations

Implementation

I will put here only code snippets, which differs from previous chapter.

Start thread: Do not forget to increase the priority and start the rendezvous AO.

```

TInt CThreadLoader::StartPeriodicalOperation( MThreadLoaderObserver* aThreadLoaderObserver )
{
    ...
    iThread.SetPriority( EPriorityMuchMore );
    ...
    err = iRAO->StartRendezvous( iThread, this );
    ...
}

```

Synchronous operation: Do not forget to wait here and issue the rendezvous request. You can see here also check for the flag, which is set when the users cancels the operation

Synchronous_operations

```

void CThreadLoader::StartPeriodicalOperationThreadTrappedL( TAny *aPtr )
{
    CThreadLoader* threadLoader = (CThreadLoader*)aPtr;

    for ( TInt i = 0; i < 1000; i++ )
    {
        // This simulates synchornous operation
        for ( TInt j = 0; j < 10; j++ )
        {
            RFs fs;
            fs.Connect();
            fs.Close();
        }

        User::After(1);
        threadLoader->iThread.Rendezvous( i );

        // Check if user did not cancel the synchronous operation
        if ( threadLoader->iPeriodicalCancelled )
            break;
    }
}

```

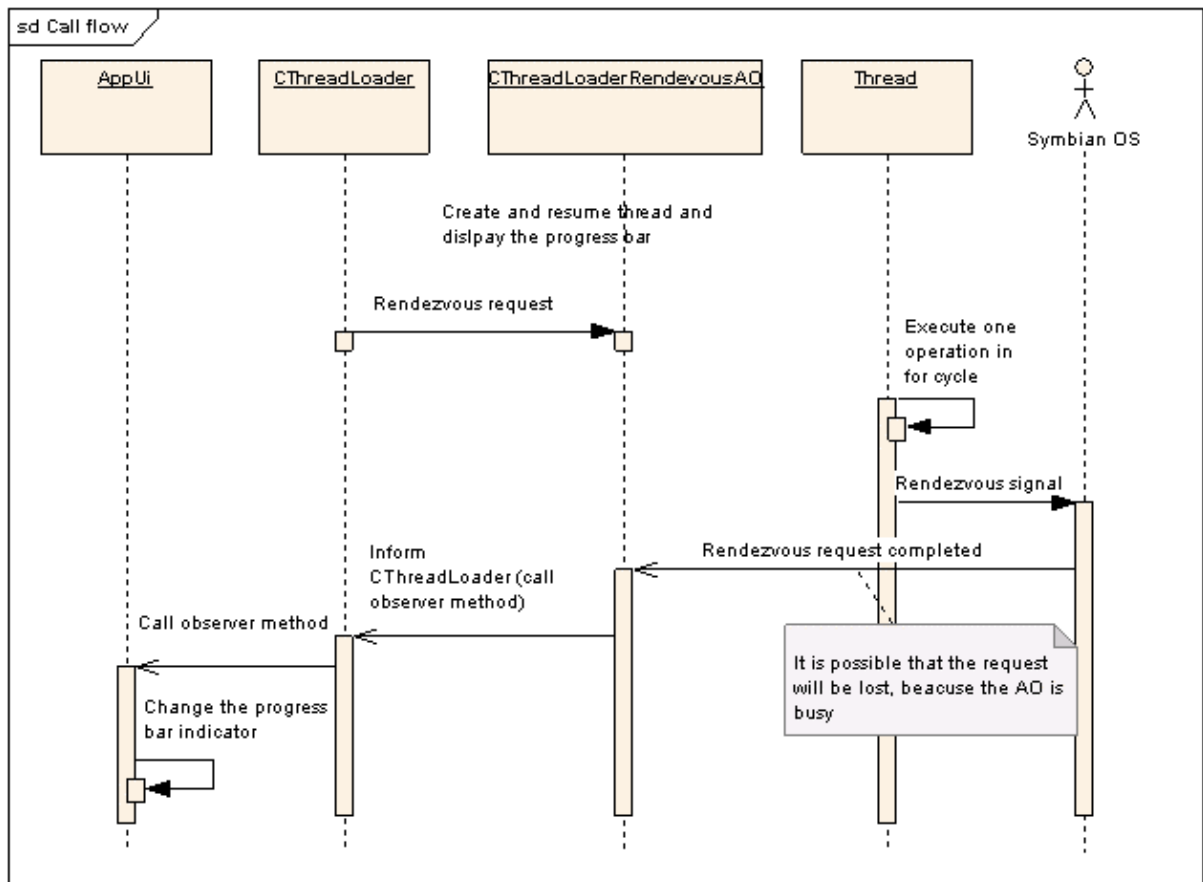


Figure 6. Call flow for periodical synchronous operations

Enhancing previous solution with message queue RMsgQueue

The rendezvous mechanism is not the best IPC we could use, the most comfortable way is use of message queue class **RMsgQueue**. In this case the queue is created and both sides opens it, while one thread writes data into it, the main UI thread reads them. This time I will not put here diagrams, as the enhancement of the code is similar to what I have did in previous paragraph. I just add one new active object.

There are the major code changes:

```
// Insert data into queue
TQueueData d;
d.iIndex = i;
threadLoader->iQueue.Send( d );

// Read data from queue and call observer
TQueueData d;
iQueue.Receive( d );
iThreadLoaderObserver->OperationIncrement( d.iIndex );

// Start watching queue for incoming data
if ( !IsActive() )
{
    if ( aQueue.Handle() != NULL )
    {
        iQueue = &aQueue;
        iQueueObserver = aQueueObserver;
        iQueue->NotifyDataAvailable( iStatus );
        SetActive();
        return KErrNone;
    }
    else
        return KErrArgument;
}
else
    return KErrInUse;
```

Example source code

Everything I presented here you can find in the attached example. It shows all 3 issues discussed here. Application was developed under S60 3rd MR SDK in CW 3.1.1. I does not use any cabability and it has assigned UID3 from unprotected range, so it can be self-signed. It was tested on N80 and N95 devices.

You can download an example with source code here: [File:FN wiki sync op src v0 1 071126.zip](#)

I welcome any comments!