OMiSCID in C++

I OMiSCID ?

O[3]MiSCID stands for Object Oriented Opensource Middleware for Service Communication Inspection and Discovery. It is also known as OMiSCID.

I.1 Basic Concepts

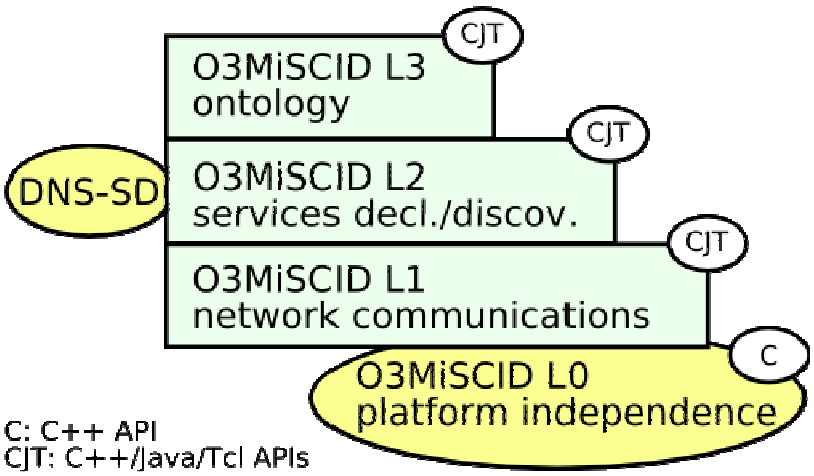
OMiSCID is a middleware dedicated to service sharing over a network. It is based on DNS-SD from apple (aka Bonjour, Rendez-vous …). It allows communication, inspection and discovery of services. It also permits to inspect these services to reach their description, the list of their variables and the inputs/outputs exposed by them. The basic concepts of OMiSCID are:

- **Services**, the fundamental concept of OMiSCID. A service is a software component which exposes a functionality over the network. To do that, it exposes a set of variables and connectors for communication. The same program can propose several services. Every service gets a unique PeerId (a 32 bits value) on the network allowing distinguishing it from the other services.
- **Variables** are simply named entities associated to values (characters strings). One can query remotely the value of a variable or request to be notified of every value change.
- **Connectors** (or channels) are communication points for the service to exchange messages (over TCP/UDP or in the future using shared memory if services are located on the same computer). A connector can be an input, an output or an inoutput (if the service can receive and send messages at the same time over this connector). Each OMiSCID service has at least one specific inoutput connector: the control connector. Using the connector, one can do inspection and control (like remote variable changes). Each connector gets a unique PeerId over the network which is a sub-PeerId from the service PeerId.
- **Connections** are links between 2 connectors. The links are always established using TCP/IP even if further communications will be done using UDP or shared memory. Doing that, when a TCP link is broken (i.e. when a service die unexpectedly), we can be aware and marks the connection as broken.

The semantic and the data associated with variables and connectors are specific to each service.

Architecture

The architecture of OMiSCID is splitted in several layers as we can see on the following figure.



On this figure, we see the various OMiSCID layers and their access language. There is initially the layer 0 (L0) which is useful only in C++. It is an abstraction of the system (Windows, Linux, Mac OSX) as the one provided by Java natively (Threads, Mutex, etc.). This layer can be used alone to make multiplatform development.

The layer 1 (L1) allows the low communications level mechanisms over the network. These communications are of 2 types: reliable or faster but non sure (with packets lost). These communications are encapsulated in a protocol called BIP (Basic Interconnection Protocol). Each message sent is atomic and is received as such by the other partner of the connection. The management of PeerIds makes possible to identify which service is at the other side of the connector.

The Layer 2 (L2) is used to make discovery, inspection and control of service. At this level, one defines the name of the service, its variables and connectors. All mechanisms, like the variable subscriptions of service searches and queries, are managed by L2. A user friendly API that looks like the Java API is provided in order to access to this layer.

The last layer (L3) allows constructing/searching services and building applications using only ontological descriptions (expressed in XML for the moment). This layer is under development and as it is not available now, we will not present it in this tutorial.

I.2 Practice

I.2.a Distributed accumulator

We will build 2 kinds of service in this example. The first one is an “Accumulator” service which gets an “Accu” variable in order to work on and an inoutput connector to receive operations as strings (+4, -5.5, /12, = 5.678…) and to send back error messaged if the command is not correct. If several clients are connected, commands will be processed in order of receiption. The accumulator source code is ‘*Example/Accumulator.cpp*’

The second service is an “Accumulator Client” that searches for an Accumulator. When one is available, subscribe to variable changes on the variable “Accu” and connects to the “Commands” connector in order to send inputs from the user. The client source code is ‘*Example/ClientAccumulator.cpp*’

I.2.b Mandatory software

To run this practice, you need *Bonjour* and *Bonjour SDK* from Apple, the xml library *libxml* and its development part, a C++ compiler (g++ or Visual Studio) and the last version of OMiSCID. You will also need the OMiSCID basic jar and GUI files with a java machine in order to make preliminary tests.

I.2.c The Accumulator service

First of all, we need to include all stuff about the user friendly interface. We need also say that we will use classes defined in OMiSCID. You can do these both steps with the following lines:

```
#include <ServiceControl/UserFriendlyAPI.h>
using namespace Omiscid;
```

Then, we will create a service using a Service Factory. This factory will provide in return a pointer on a newly created service. The creation of the service using the factory does not register it using Bonjour. This step is simply done with this piece of code:

```
Service * pAccuServer = ServiceFactory.Create( "Accumulator" );
```

We only provide a name for the service. We may also add a second parameter to provide a class. Here, the default class value for the "Accumulator" is "Service".

Now, we will add to the service a variable called "Accu" which is a float, with a specific description and a ReadAccess (remote accesses are only read). Then, we just put the default value of the variable to 0.0.

```
pAccuServer->AddVariable( "Accu","float","Accumulator value",ReadAccess );
pAccuServer->SetVariableValue( "Accu", 0.0f );
```

Now, we add an inoutput connector "Commands" with its description.

```
pAccuServer->AddConnector("Commands","Input for commands. output of errors.",
    AnInOutput );
```

And finally, we start the service, i.e. we register it and start its control port.

```
AccuServer->Start();
```

As the service is using threads to run, we just have to wait forever. So, we just create an event and wait for it without any delay. As no one will signal this event, we will wait forever:

```
Event Forever;
Forever.Wait();
```

Now, the service is ready. You can use the GUI (using the 'java –jar omiscidGui.jar' command), to inspect it. You can see the "Accu" variable and the "Commands" connector. You can also try to send commands on the connector. You may see that nothing appears… Why? Even, if the commands are correctly received, we do not say to OMiSCID what have to be done with them. They are simply dropped.

We need to subclass the ConnectorListenner class and mainly its MessageReceived method in order to handle data coming from the connector. This method has 3 parameters. The first one is a reference to the service which received this message. The second parameter is the name of the incoming connector. The last one is the message data. This is the class declaration of our AccumulatorConnectorListener:

```
class AccumulatorConnectorListener : public ConnectorListener
{
public:
	/* @brief constructor */
	AccumulatorConnectorListener();
	/* @brief destructor always virtual */
	virtual ~AccumulatorConnectorListener();

	/* @brief callback function to receive data */
	Virtual void MessageReceived(Service& TheService, const SimpleString
LocalConnectorName, const Message& Msg);

private:
	Mutex Locker;		/*!< Lock access to my variable */
	float Accu;			/*!< my accumulator */
};
```

We can see here that we only override the MessageReceived method. We also use a Mutex to protect access to our Accu value. So first, here are the constructor/destructor codes.

```
	/* @brief constructor */
AccumulatorConnectorListener::AccumulatorConnectorListener()
{
	// set initial value to the Accu
	Accu = 0.0f;
}

	/* @brief destructor */
AccumulatorConnectorListener::~AccumulatorConnectorListener()
{
}
```

Note that you should always turn the destructors as virtual. One can see nothing special here. We just initialised the Accu member to 0.0 into the constructor. Now let's take a look at the MessageReceived implementation.

```
	/* @brief callback function to receive data */
void AccumulatorConnectorListener::MessageReceived(Service& TheService, const
SimpleString LocalConnectorName, const Message& Msg)
{
	// Error management
	bool IsWrong = true;	// by default, the command is wrong
	SimpleString ErrorMessage = "Bad command.";

	// Other data declaration here
	[…]

	// Start lock myself using my mutex (needed if multiple clients)
	Locker.EnterMutex();

	// Get the pointer to the data and parse data
	Command = Msg.GetBuffer();

	// Here the code that parses the data, it is not an issue of this tutorial
	[…]

	// At the end, if the command is ok
	if ( IsWrong == false )
	{
		// export the new Accu value, so the clients will receive notification
		TheService.SetVariableValue("Accu", Accu);
	}
```

```
	else
	{
		// Discard value and send back to the rigth PeerId an error message
		// in safe mode (TCP or SHM - when implemented)
TheService.SendToOneClient( LocalConnectorName, (char*)ErrorMessage.GetStr(),
			ErrorMessage.GetLength(), Msg.GetPeerId(), false );
	}

	// Unlock myself
	Locker.LeaveMutex();
}
```

As we saw, receiving data and responding is somehow easy. Now, we must change the main program to reflect the usage of this new class. At the beginning, create an AccumulatorConnectorListener and after creating the "Commands" connector, let's say that we want to use this object to receive messages.

```
// Instanciate a Connector listener (must *not* be destroyed before it has
// been removed for the connector or until the service is destroyed).
AccumulatorConnectorListener MyCommandsListener;

// Some code here…
[…]

// Add a Connector to receive and send messages (AnInOutput) and set my callback
// object to receive notification of messages
pAccuServer->AddConnector( "Commands", "Input for commands. output of errors.",
                           AnInOutput );
pAccuServer->AddConnectorListener( "Commands", &MyCommandsListener );
```

I.2.d Searching, connecting and communicating with a service: the Accumulator Client

We will now build the client service for the accumulator. This service needs to find an Accumulator with a variable "Accu" and an InOutput connector "Commands". Then, it must connect to him and ask to receive data coming from the connector and variable change notifications. So, we will first built a class that subclasses both ConnectorListener and RemoteVariableChangeListener.

```
class ClientConnectorAndVariableListener : public ConnectorListener, public
RemoteVariableChangeListener
{
public:
	/* @brief constructor */
	ClientConnectorAndVariableListener()
	{
	}
	/* @brief destructor */
	virtual ~ClientConnectorAndVariableListener()
	{
	}

	/* @brief callback function overide to receive data */
	void MessageReceived(Service& TheService, const SimpleString
LocalConnectorName, const Message& Msg)
	{
		// Create a SimpleString with message and output it
		// even if we can directly output it. Show usage of SimpleString

		SimpleString Message = Msg.GetBuffer();
		cerr << Message << endl;
	}
```

```
	/* @ brief callback for variable changes notification */
	void VariableChanged(ServiceProxy& SP, const SimpleString VarName, const
SimpleString NewValue )
	{
		cout << "Current Accu value: " << NewValue << endl;
	};
};
```

In this class, we only print messages coming from the Accumulator and print new value of the variable when it is modified. Now, let's start our client service in the main function. We first instantiate a ClientConnectorAndVariableListener object and ask the registry to create a new service. Then, we add a connector and register our connector listener.

```
	// Instanciate a Connector and a Variable listener (must *not* be destroyed
	// before it has
	// been removed for the connector or until the service is destroyed).
	ClientConnectorAndVariableListener MyListener;


	// Ask to the service factory to create a Service. The service is not
	// register yet. We do not provide the service class, the default value
	// 'Service' will be used
	Service * pAccuClient = ServiceFactory.Create( "Accumulator Client" );

	// some check here…

	// Add a Connector to send commands and receive error messages (AnInOutput)
	// and set my callback
	// object to receive notification of messages
	pAccuClient->AddConnector( "SendCommands", "A way to send commands to the
                                  Accumulator", AnInOutput );
	pAccuClient->AddConnectorListener( "SendCommands", &MyListener );
```

We will use the newly created service in order to search for an Accumulator. We loop several times on a search for 5 seconds in order to give feedback to the user. Obviously, we can ask for a longer timeout and call only once the search. The result of a single search is a ServiceProxy.

```
// A proxy to communicate with the Accumulator
ServiceProxy * OneAccumulator = NULL;
// Loop for searching an Accumulator
printf( "Search for an Accumulator service.\n" );
for( int NbTry = 1; NbTry <= 5; NbTry ++ )
{
	// Search for services, wait 5 s (5000 ms) to get an answer
	// Get in return a ServiceProxy*. This service is nammed
	// Accumulator *AND* gets a variable
	// Accu *AND* an InOutput connector "Commands"
	OneAccumulator = pAccuClient->FindService(
		And(
			NameIs("Accumulator"),
			 HasVariable("Accu"),
			 HasConnector("Commands",AnInOutput)
			), 5000 );

	if ( OneAccumulator == NULL )
	{
fprintf( stderr, "Search for an Accumulator service failed. Try again...\n" );
	continue;
	}
	break;
}
```

After the loop, if OneAccumulator is not NULL, we found what we are looking for. We can start our client service, try to connect our local connector to the “Commands” of the Accumulator, register our listener as a variable listener on “Accu” and loop on user input.

```
// register the Client service and launch everything
// we must do it before communicating with the Accumalator
pAccuClient->Start();

// Connect to the Accumulator
if ( pAccuClient->ConnectTo( "SendCommands", OneAccumulator, "Commands" ) == true )
{
	// Subscribe to variables changes for Accu. We will receive a first callback
	// when done
	OneAccumulator->AddRemoteVariableChangeListener( "Accu", &MyListener );

	// Enter command prompt, 1er time with help
	cout << "Enter new command ('+4', '/5.5', ' *8.59', '-6', ' = 8.2'...): ";
	cout << endl;

	// Do my work
	SimpleString LocalCommand;
	for(;;)
	{
		// Get the local command at keyboard
		cin >> LocalCommand;

		// Send to all connected Client... Here we are connected to only one
		// accumulator we can also use SentToOneClient...
		pAccuClient->SendToAllClients( "SendCommands",
			(char*)LocalCommand.GetStr(), LocalCommand.GetLength(), false );
	}
}
else
{
	fprintf( stderr, "Could not connect to the Accumulator connector. quit !\n"
);
```

You can try to launch multiple clients over a single Accumulator to see what happens.

I.2.e Registering many services, multiple searches paradigm

You may want to search for several services at the same time. You can obviously call many times FindService but the best way is to call FindServices. You can refer to ‘Examples/RegisterSearchTest.cpp’ to learn how to do that.