MSR Networked Embedded Sensing Toolkit
(MSR Sense v0.1.x)
Programmer's Guide

July 2006

This document describes the software architecture of the MSR Sense service runtime (µSEE) and instructions on how to extend service libraries. The default development language is C# on .NET 2.0 and the default development environment is Visual Studio 2005.

1. Introduction

A µSEE provides a way to compose loosely coupled software component, called *services*, at run time. Services have *ports* and *parameters*, which are their only interface to the rest of the system. Ports can be either input or output. Services can be configured by setting their parameters and can be composed by connecting the ports.

A typical workflow of µSEE is as following:

- µSEE is started with an initial configuration
- µSEE receives user a user task specified in MSTML tasking markup.
- µSEE parses the MSTML specification and load services from service libraries and compose them together.
- Services are executed until µSEE receives a `wrapup` command from the user.
- µSEE configuration can be changed when there is no task running.

The rest of this section gives an overview of the above steps.

1.1 µSEE Configuration

A µSEE is typically started as a command line process, such as

```
> miusee –ini localhost.ini.xml
```

where the ini file specifies the initial configuration of the microserver, such as what serial forwarder to listen to, what mote ID to expect, and what service libraries to load.

Here is an example of the ini file:

```
<?xml version="1.0" encoding="utf-8" standalone="yes" ?>
<!--This is an example ini file for miuSEE.-->
<Config MSTMLPort="6000" PortMin="65000">
  <SerialForwarders>
    <Forwarder HostName="localhost" PortNumber="9001" />
  </SerialForwarders>
  <Motes>
    <Mote>
      <MoteID>-1</MoteID>
      <AMHandler>10</AMHandler>
    </Mote>
  </Motes>
  <Services>
```

```
    <Assembly>Microsoft.NEComp.Microserver.Library</Assembly>
    <Assembly>Microsoft.NEComp.Microserver.Library2</Assembly>
  </Services>
</Config>
```

In this example, µSEE is started to expect MSTML tasks from the port number 6000 (`MSTMLPort`). If the microserver needs to serve socket connections from other devices, it will user ports above 65000 (`PortMin`) for socket servers. The rest of the configuration file has three sections:

- `SerialForwarders` section specifies a set of serial forwarders that the microserver will connect to. Each serial forwarder is specified as a hostname and port number. Each serial forwarder (or a MoteForwarder in MSR Sense releases) is a socket server and mSEE will connect to it as a socket client.
- `Motes` section specifies a set of mote ID and active messaging handler (AMHandler) addresses that the microserver should expect to receive. Serial forwarders can pick up any mote packets it can hear. This specification allows microserver to filter out certain packets it does not want to receive. The -1 in the setting is a wild card. For example,
  ```
  <MoteID>-1</MoteID>
  <AMHandler>10</AMHandler>
  ```
  tells the microserver to expect packets with AMHander 10 from any mote.
- `Services` section specifies the service libraries to load. Each of them is called an *assembly*. Their names are the namespaces of the C# assemblies.

1.2 MSTML

There are two ways for a microserver to receive tasks. A task can be specified as a command line argument. For example,

```
> miusee -ini localhost.ini.xml -mstml mytask.xml
```

commands the µSEE to load `mytask.xml` from local disk after it loads the configuration.

The second way is to send an MSTML file over the network to the MSTMLPort.

Here is an example of MSTML file that specifies a service composition as shown in Figure 1.

```
<?xml version="1.0" standalone="yes"?>
  <entity name="OscilloscopeApp">
    <port name="mote" type="AMHandler">
      <property name="input"/>
      <property address="-1:10"/>
    </port>
    <port name="port1" type="Socket">
      <property name="output"/>
      <property address="localhost:5000"/>
    </port>
    <entity name="TOSReceiver" type="ComplexTOSPacketReceiver">
      <property name="messageType" value="ArrayOscopeMsg"/>
    </entity>
    <entity name="ToXML" type="DataToXML">
    </entity>
    <relation name="relation1"/>
```

```
  <relation name="relation2"/>
  <relation name="relation3"/>
  <link port="mote1" relation="relation1"/>
  <link port="TOSReceiver.Input" relation="relation1"/>
  <link port="TOSReceiver.Output" relation="relation2"/>
  <link port="ToXML.Input" relation="relation2"/>
  <link port="ToXML.Output" relation="relation3"/>
  <link port="port1" relation="relation3"/>
</entity>
```

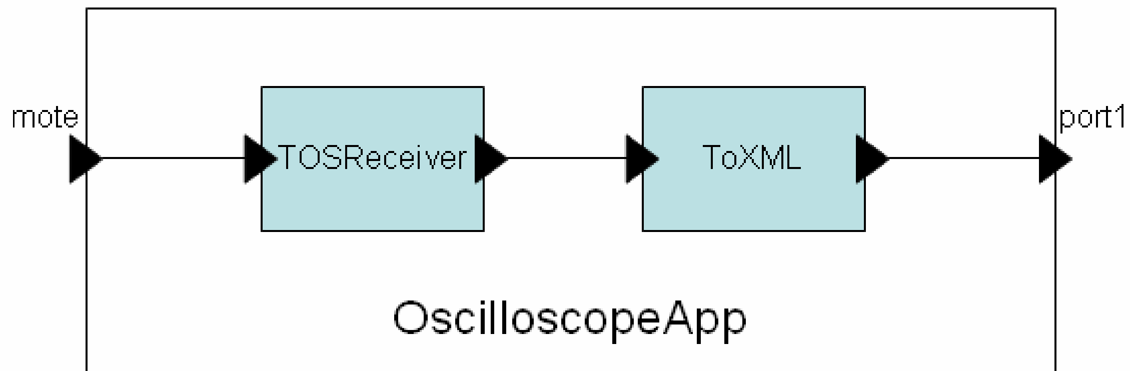

Figure1. A service composition.

MSTML is modified from MoML (Modeling Markup Language). This example specifies a top-level Entity which has name OscilloscopeApp. It further contains 4 sections:

- External ports: these are ports that communicate with other devices. There are primarily two types: mote connections and socket connections. Each of them can be either input or output. In this example, there is a mote input port, which receives data addressed to any mote with AM handler 10. There is an output port connected to a PC (local host). The PC should setup a socket server at port 5000 to receive data from this microserver.
- Services: they are labeled as entities in MSTML. They must have a unique name (within the top level entity) and a service type. Service type is the class name of the service in a service library. A service may have parameters, which are properties of the entity. A parameter is identified by its name and value. For example
  `<property name="messageType" value="ArrayOscopeMsg"/>`
  specifies that the parameter with name "messageType" is set to a string value "`ArrayOscopeMsg"`.
- Relations: they are mediators for connections. In order to support connections among multiple input ports and multiple output ports (i.e. fan-in and fan-out), they are explicitly spelled out in MSTML.
- Links: they specify how each port of each service is connected to a relation. There is a link between exactly one port and one relation. External ports are represented by their names, while a port of a service is represented as ServiceName.PortName. Note that a unconnected input port will never receiver data, and a unconnected output port will silently discard any data it suppose to send.

After μSEE receives/loads a MSTML, it parses the file. For each mote port, it creates a MotePacketDispatcher. For each output socket port, it tries to connect to the socket server stated at the specified address. For each service, it finds the service in the libraries and instantiate it with the specified name. It then finds the parameter with the corresponding name and sets its value. It creates a relation object for each relation and links the ports to them.

1.3 Service Execution

Each service is executed in a separate thread in μSEE. The reason is to better support run-time service reuse. The communication between the ports has a publish/subscribe semantics. That is, each output port is a data publisher, each relation is a mediator and each connected input port received a copy of the data.

The executions of services are event-triggered. The thread is idle by default and only waked up by input data (called tokens). It is up to the service implementation to decide how to respond to the input tokens. In particular, the μSEE does not provide any synchronization mechanism.

2 μSEE Kernel

The kernel of mSEE is in the Microsoft.NEComp.Microserver package in the directory of %MSRSENSE%\miuSEE\Microserver. Figure 2 shows a UML diagram for key classes in mSEE kernel.

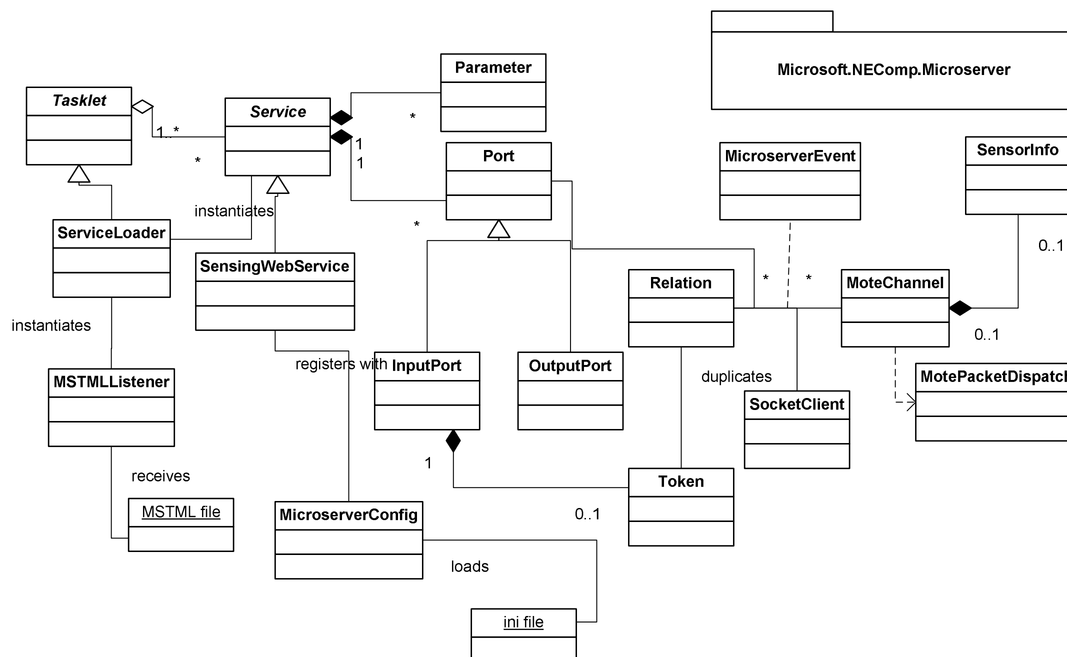

Figure 2. A UML diagram for mSEE kernel.

In UML static class diagrams, boxes with labels are classes; links with triangles are class inheritance relations (with the triangle side indicating the supper class); links with diamonds are containment relationships. So this diagram illustrates:

- Services are contained by Tasklet. A tasklet is a portion of a task which runs on a single microserver. Recall that in our design a task may span across multiple microservers. A service can be contained by multiple tasklets, and a tasklet can also contain multiple services.
- A service contains a number of ports and parameters. Each port or parameter can only be contained by one service.
- A port can be an input port or an output port. In μSEE, a port cannot be both input and output at the same time.
- Ports, MoteChannels, and SocketClients can be connected via relations. A relation implements the publish/subscribe semantics with the help of an event class called MicroserverEvent.
- Input ports do not directly contain any token. We use the system event pool and the delegate model to implement publish/subscribe semantics. Tokens FIFO ordered and are duplicated by relations in a nondeterministic order if there are more than one input ports.
- ServiceLoader implements tasklets, so they contain services.
- A MSTMLListener receives MSTML files and instantiate a ServiceLoader to parse the file. The service loader instantiates services and relations according to the MSTML file.
- The MicroserverConfig class loads and parses an ini file and instantiates SocketClients and MoteChannels.

2.1 Service Base Class

The Service class is of key interests in this document in order to further describe how to extend service libraries. Service is an abstract class that defines the following methods.

2.1.1 Containers and Constructor

A service must be contained by one or more containers that extend the Tasklet abstract class. A typical container is a ServiceLoader. Each service maintains a list of containers. When a service is constructed, it must know which tasklet is using it. It then adds the tasklet into the list. When a task is finished, the corresponding tasklet is removed from the list. Every service executes in its own thread. When the container list is empty, meaning that the service is no longer in use, the service kills the thread and hands itself over to the garbage collector. **Note that the multi-container feature has not been fully developed. There can only be one active ServiceLoader in this release.**

Service executions are triggered by receiving tokens in their input ports. In order to preserve atomicity in service execution, there is a unique lock (called fireLock) that the service must grab before it can proceed with reacting to a token. By default, the firelock object is the ServiceLoader that instantiate the service. (Note, obviously, in a true multi-container scenario, the fireLock should be a singleton object rather than the separate service loaders).

Thus, a service has the following constructor:

```
public Service(string name, Tasklet container, object fireLock);
```

The service name must be unique within the tasklet.

2.1.2 Ports and Parameters

A service may contain a number of ports and parameters. A port can be either an input port or an output port, but not both. The service maintains a list for ports and a list for parameters. More precisely, they are hash tables indexed by the name of the ports and parameters. The names of ports and parameters must be unique within the service.

Ports and parameters have the following constructors:

```
public InputPort(Service service, string portName);
public OutputPort(Service service, string portName);
public Parameter(Service service, string parameterName, object val);
```

Once ports and parameters are created, they are automatically added to the lists in the container service.

When constructing a parameter, one must give an initial value `val`. Note that the type of the initial value determines the type of the parameter. Further values of the parameter must be convertible to the initial type. For example, if a parameter is going to be used as a double, the initial value should be set to something like 0.0 rather than integer 0. Currently, the type of the parameter can only be primary types such as numerical types and a string.

2.1.3 Service Execution

A service owns its own thread. The thread is attached to the Run() method of the service. In the default implementation, the Run() method first calls the Initialize() method, and then waits on the input ports. If there is a token received at any input port, the thread is waked up. It then sequentially checks which input port has a token and calls the Fire() method to process that token. The service monitors its container list. A container is removed (by ServiceLoader) by calling the RemoveContainer() method of the service. When the list is empty, the service stops the thread and calls its own Wrapup() method. The service should clean up any resources it occupied.

The execution methods have the following signature defined in the Service base class.

```
public virtual void Initialize()
abstract public void Fire(Port p, Token t)
public virtual void Wrapup()
```

The default behavior of Initialize() and Wrapup() are implemented in the base class, and Fire() must be implemented by extended classes. The default behavior of Initialize() is to do nothing, and the default behavior of Wrapup() is to disconnect all its ports. When Fire() is called in the Run() method, it receives as arguments which port triggers the fire and what token is contained in the port. Desingers of concrete services should implement these methods to achieve desired behaviors.

3 Extending Service Libraries

In this section, we discuss how to implement new services through an example. Suppose that we want to add a new service called FIRFilter to a service library called LibraryX.

A FIRFilter has one input stream, x, one output stream, y, and computes at the n-th firing:

$$y_n = a_0 \cdot x_n + a_1 \cdot x_{n-1} + \ldots + a_k \cdot x_{n-k}$$

where $a_0, \ldots a_k$ are called the tabs. The tabs are the parameters for the service.

3.1 Set up

To implement the service, we start up Visual Studio 2005 and create a new Class Library project called LibraryX. Rename the default file created by Visual Studio from Class1.cs to FIRFilter.cs.

In the solution explorer, right click on the References line in the project tree and select "Add Reference". An Add Reference dialog box will pop up. In the "Browse" tab, select the Microsoft.NEComp.Microserver.dll in your MSR Sense installation -- typically in %MSRSENSE%\bin.

Add Microserver kernel packet to the using list:

```
using Microsoft.NEComp.Microserver;
```

Modify the generated class signature to extend the Service base class. So we get a skeleton like:

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.NEComp.Microserver;

namespace LibraryX {
    public class FIRFilter : Service{

    }
}
```

Next we will fill up the class by providing the concrete implmentation of the constructor and execution methods.

3.2 Port and Parameters

Ports and parameters are public members of the class:

```
        public Port input;
        public Port output;
        public Parameter tabs;
```

We also create private variables that will be used in future execution:

```
        private double[] tabValues;
        private List<double> pastInputs;
```

3.3 Constructor

The constructor should create the input and output ports and the parameter called tabs.

```
        public FIRFilter(string name, Tasklet container,
            object fireLock) : base(name, container, fireLock) {
```

```
        input = new InputPort(this, "input");
        output = new OutputPort(this, "output");
        tabs = new Parameter(this, "tabs", "0.5, 0.5");
    }
```
So the “tabs” is implied to be a string, containing comma separated double numbers.

3.4 Initialization

The initialize method is called at the beginning of the execution lift cycle. It is the first place where the parameters of the service can be updated. So the main job of Initialize() in this service is to set up memory spaces for storing inputs and to parse tab values.

```
    public override void Initialize() {
        base.Initialize();
        String[] tabStrings = ((String)tabs.Value).Split(",");
        int n = tabStrings.Length;
        if (n == 0) {
            tabValues = null;
        } else {
            tabValues = new double[n];
            pastInputs = new List<double>(n);
            for (int i = 0; i < n; i++) {
                tabValues[i] = Double.Parse(tabStrings[i]);
                pastInputs[i] = 0.0;
            }
        }
    }
```
In this implementation, the tabs parameter string is parsed and converted to an array of doubles. The input (and its limited history values) are stored in the pastInputs list.

3.5 Fire

The Fire() responds to the inputs and computs the outputs according to the filtering fomula. An example of the implementation is

```
    public override void Fire(Port p, Token t) {
        double yn = 0;
        int n;
        if (tabValues == null) {
            n = 0;
        } else {
            n = tabValues.Length;
        }
        if (n != 0) {
            double xn = (double)t.getData();
            pastInputs.RemoveAt(n-1);
            pastInputs.Insert(0, xn);

            for (int i = 0; i < n; i++ ) {
                yn = yn + tabValues[i]*pastInputs[i];
            }
        }
        output.Put(new Token(yn));
    }
```
Since this service only has one input port, we do not have to test where the token is received. `Token.getData()` is to get the payload data in the receiving toke.

`output.Put(Token)` is called to send the token from the output port to downstream services. The rest of the method is just book keeping and the arithmetic.

3.6 Compilation

Before compiling the code, you may want to decide where you put the dll. We suggest that all dll files go into the %MSRSENSE%\bin directory. To do this, in the Solution Explorer right click on the LibraryX project and select Properties. In the Build tab, find the Output path text box and type in your %MSRSENSE%\bin path name. Build the solution by pressing Shift+F6.