InstantIO Manual

Patrick Dähne, Patrick.Daehne@igd.fraunhofer.de, 2008-03-18

Introduction

This document describes the InstantIO framework. The main task of this framework is to efficiently handle input and output devices. But besides simply allowing access to devices, the InstantIO framework also handles all preprocessing of the data provided by the devices. For example, it does not only allow the application to grab video frames from a web cam, but also to determine the position and orientation of the user by using these video frames (i.e. video base tracking).

The intention is to create a library of small software modules that are specialized on simple tasks. Some software modules act as device drivers, i.e. they produce or consume data streams. Others act as filters, i.e. they transform incoming data streams. The modules are nodes of a data flow graph, and they receive data from and send data to other modules via the edges of the graph. The application developer should be able to build as much as possible of his application by simply assembling these prefabricated software modules into a data flow graph, allowing him to concentrate on the application logic instead of being stuck in the arduous task of creating the basic infrastructure of his MR application.

It is possible to create the data flow graph by using a C++ or Java API, but this is not the recommended proceeding because changing the graph requires to modify and recompile the application. Instead, a special XML dialect has been developed that allows to store the structure of the graph in a configuration file. When starting the application, the framework automatically restores the graph from such a configuration file.

The configuration files can be edited by using a text editor. Another alternative is to create the graph on the fly by using the integrated graphical user interface (GUI). The GUI allows to completely modify the whole graph during the runtime of the application, and it is able to save the current structure of the graph into a configuration file.

Unfortunately, when implementing MR applications in reality, it is usually not possible to stick in all cases to this concept of a data flow graph. There are older, legacy software components from previous projects that have to be ported to the new application. Or software components from other project partners have to be integrated that are using a completely different software architecture. Adjusting such components to a data flow graph is a tedious task that usually requires writing ugly “wrapper” nodes that handle the communication between software component and data flow graph.

To prevent the necessity of writing wrapper nodes, the InstantIO software API is split into two layers: A low-level interface, and a high-level interface.

The low-level interface is the communication layer of the device management system. It provides thread-safe communication channels between software components. Referring to the concept of a data flow graph, the low-level interface consists of the edges between the nodes of the graph.

The high-level interface on the other hand allows building data flow graphs consisting of nodes that are state machines, receive data values on incoming edges, change their state depending on the incoming data, and send data values on their outgoing edges. The high-level interface is build

on top of the low-level interface.

This subdivision of the system into different layers makes it easy to integrate software components that are not itself nodes of the data flow graph. Instead of writing wrapper nodes, these components use the low-level interface to “inject” events into the scene graph.

Low-level interface and high-level interface are APIs provided by C++ libraries or Java packages. These APIs are usually only relevant when integrating legacy software components into the system, or when implementing new nodes of the data flow graph. Applications are allowed to use these APIs to construct the data flow graph, but as mentioned before this is not the recommended approach because doing so requires to modify and recompile the application when the data flow graph needs to be changed. Instead, the preferred way to build the data flow graph when starting the application is to parse a configuration file.

An important feature of the framework is that it allows the data flow graph to span different machines on the network. MR applications quite often have to handle devices that are not connected to the machine the application is running on, e.g. stationary tracking systems that determine the position and orientation of a mobile device. Furthermore, they often must be able to allow the user to communicate with other users (computer supported cooperative work), or they need to receive data from stationary servers. The InstantIO framework allows the application to communicate seamlessly other system components, no matter where they are located on the network.

The following sections describe different aspects of the framework in more detail.

Low-level Interface

The low-level interface is the communication layer of the device management system. It provides thread-safe communication channels used to exchange data between different software components.

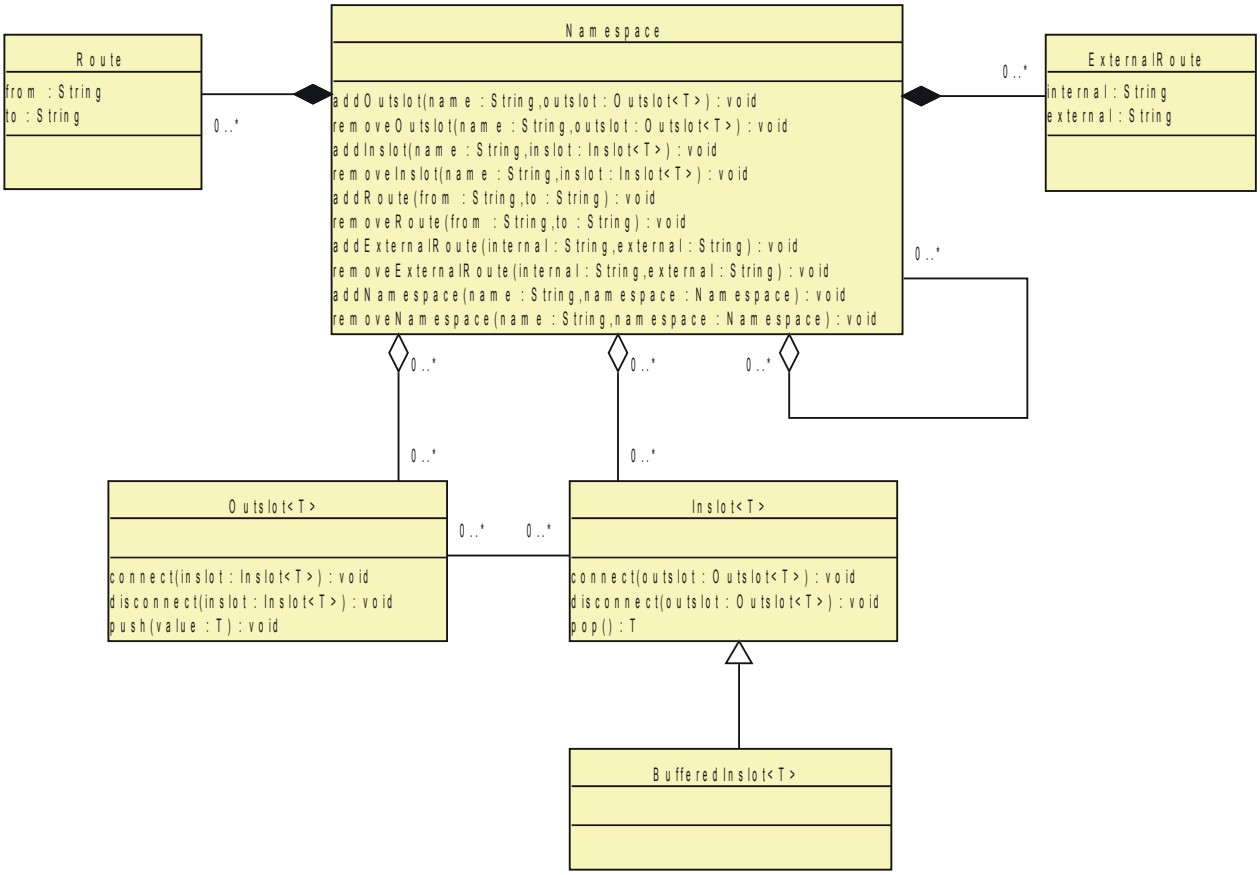

Figure 1: UML diagram of the lowlevel interface

OutSlots and InSlots

OutSlots and InSlots are the basic communication primitives. OutSlots are used to send data to other software components, and InSlots are used to receive data from other software components. Both OutSlots and InSlots are typed, i.e. you have to specify what kind of data can be sent to or received from the slot when you create it. In C++, the data type is simply a template parameter of the template OutSlot and InSlot classes, in Java, you have to specify the data type by providing the runtime class of the data type.

The following code examples demonstrate how to create an InSlot used to receive boolean data values:

C++

```
#include <InstantIO/InSlot.h>

using namespace InstantIO;

InSlot<bool> myInSlot;
```

Java
```
import org.instantreality.InstantIO;

InSlot myInSlot = new InSlot(Boolean.class);
```

Creating a corresponding OutSlot used to send boolean data values looks like this:

C++
```
#include <InstantIO/OutSlot.h>

using namespace InstantIO;

OutSlot<bool> myOutSlot;
```

Java
```
import org.instantreality.InstantIO.*;

OutSlot myOutSlot = new OutSlot(Boolean.class);
```

OutSlots and InSlots can be connected when they are using the same data type. This connection can be created manually, as shown in the following code example. Here, we connect the OutSlot and the InSlot we have just created in the examples before:

C++
```
myOutSlot.connect(&myInSlot);
```

Java
```
myOutSlot.connect(myInSlot);
```

In this example, we connect the OutSlot to the InSlot. It is also possible to do it the other way round, i.e. to connect the InSlot to the OutSlot, as shown in the following example. Both ways to interconnect slots are completely equivalent.

C++
```
myInSlot.connect(&myOutSlot);
```

Java
```
myInSlot.connect(myOutSlot);
```

As already mentioned before, it is only possible to connect slots that have the same data type. Trying to connect slots of different data types will not result in an error condition, instead the *connect()* method silently fails.

Even though it is possible to connect InSlots and OutSlots manually, this is only done in very rare circumstances. To establish a manual connection, you have to know the slots at compile time, but usually, you have to create connections between slots of different software components that are not known at compile time. For this reason, there are means to connect OutSlots and InSlots to each other automatically by following the rules established by the application developer. We will explain this mechanism later on in the following subsection.

Of course, it is also possible to disconnect OutSlots and InSlots manually:

C++

```
myOutSlot.disconnect(&myInSlot);
```

Java

```
myOutSlot.disconnect(myInSlot);
```

Instead of disconnecting the InSlot from the OutSlot, you can also disconnect the OutSlot from the InSlot – both ways are completely equivalent:

C++

```
myInSlot.disconnect(&myOutSlot);
```

Java

```
myInSlot.disconnect(myOutSlot);
```

OutSlots can be connected to an arbitrary number of InSlots. When you write a data value into an OutSlot, it transfers this data value to all InSlots it is currently connected to. When the OutSlot is not connected to any InSlot, the data value simply gets lost. Likewise, InSlots can be connected to an arbitrary number of OutSlots. InSlots receive data values from all OutSlots they are currently connected to. It is not possible to determine from which OutSlot a data value actually came from.

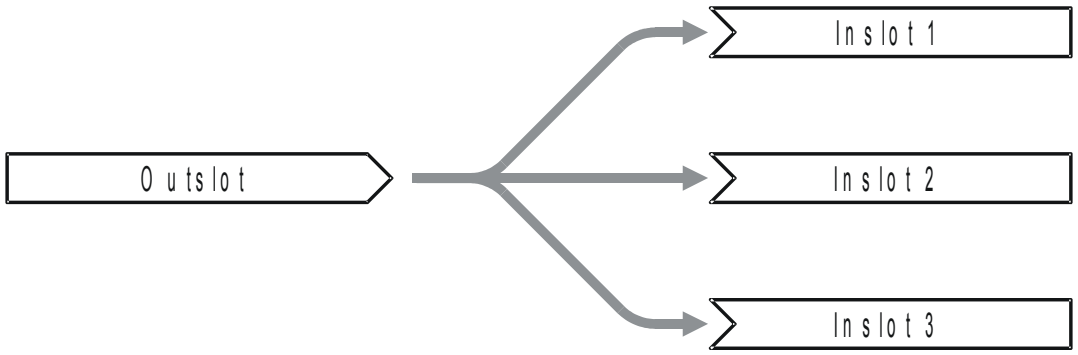

Figure 2: OutSlots can be connected to more than one InSlot. Data values written into an OutSlot are copied to all InSlots connected to that OutSlot

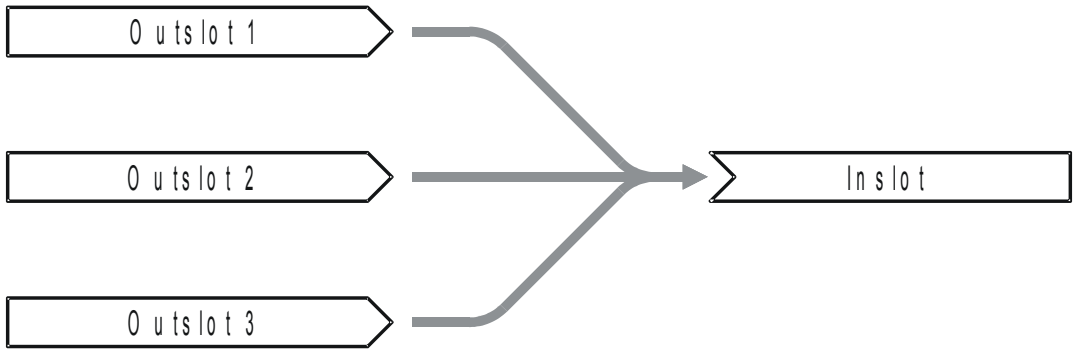

Figure 3: InSlots can be connected to more than one OutSlot. They receive data values written into any of the

connected OutSlots.

It is also possible to interconnect the same OutSlot and InSlot more than once. In this case, when you write a data value in the OutSlot, it is nevertheless transferred only once from the OutSlot to the InSlot. But when you connect two slots n times, you also have to disconnect them n times before they actually get disconnected. Trying to disconnect slots that are not interconnected does not result in an error condition, instead the *disconnect()* silently fails.

Writing data values into OutSlots to send them to other software components is straightforward. You simply push the data values into the OutSlot. In the following example, we send a boolean data value to all InSlots connected to our OutSlot:

C++
```
bool value = …; // true or false
myOutSlot.push(value);
```
Java
```
Boolean value = new Boolean(…); // true or false
myOutSlot.push(value);
```

When a software component writes data values into an OutSlot, the OutSlot automatically transfers a copy of these data values to all InSlots it is currently connected to. Data values can be given a time stamp, but the OutSlot automatically provides them with a time stamp (the current system time) when the application does not give them a time stamp. To attach an application-specific time stamp to a data value, the InstantIO *Data* object is used. The Data object combines a data value with a time stamp. The following code example demonstrates how to use the Data object:

C++
```
bool value = …; // true or false
unsigned long timeStamp = …;
Data<bool> data;
data.setValue(value);
data.setTimeStamp(timeStamp);
myOutSlot.push(data);
```
Java
```
Boolean value = new Boolean(…); // true or false
long timeStamp = …;
Data data = new Data();
data.setValue(value);
data.setTimeStamp(timeStamp);
myOutSlot.push(data);
```

It is common practice that the time stamp is specified in milliseconds since midnight January 1., 1970 UTC. Nevertheless, the time stamp is not used by the InstantIO system in any way, it is simply transferred untouched to the InSlots. So you can actually transfer any kind of data in the time stamp.

The last data value written to an OutSlot is stored in the OutSlot and can be accessed using the *getValue()* method. This can be used for optimization purposes, i.e. to prevent sending duplicate data values. Consider for example a tracking system that calculates the current position ten times per second. When you do not move, the tracking system always determines the same position

(well, in practice this will never happen due to jitter, but this is just an example). It does not make much sense to send the same position value to other software components every 100 milliseconds, because it is just a waste of system resources. For this reason, you can check if the position actually changed:

C++
```
Vec3f newValue = …;
if (newValue != static_cast<Vec3f>(myOutSlot.getValue()))
  myOutSlot.push(newValue);
```
Java
```
Vec3f newValue = new Vec3f(…);
if (newValue.equals(myOutSlot.getValue().getValue()))
  myOutSlot.push(newValue);
```

(*myOutSlot.getValue().getValue()* in the Java example is not a typo, *myOutSlot.getValue()* returns a *Data* object that combines a data value with a time stamp. To get the actual data value, you have to call the *getValue()* method of the *Data* object. For the same reason, you need that ugly *static_cast* in the C++ example.)

Another possibility to optimize the performance of your code is to check if there are actually InSlots connected to the OutSlot before you actually generate new data values. Sometimes, generating data values is quite a time-consuming task. Consider for example a video tracking system that has to determine the position of objects from video images. When there is no one interested in getting the position values (i.e. when there is no InSlot connected to the corresponding OutSlot), you can skip the whole position estimation and save system resources. To check if there are InSlots connected to the OutSlot, you can call the *isConnected()* method:

C++
```
if (myOutSlot.isConnected())
{
  // Time consuming calculation of value
  Vec3f value = …;
  myOutSlot.push(value);
}
```
Java
```
if (myOutSlot.isConnected())
{
  // Time consuming calculation of value
  Vec3f value = new Vec3f(…);
  myOutSlot.push(value);
}
```

The counterpiece of OutSlots are InSlots. InSlots receive copies of data values written to any OutSlots they are currently connected to. Getting new data values from InSlots is straightforward – you simply call the *pop()* method:

C++
```
bool value = myInSlot.pop();
```
Java
```
Boolean value = (Boolean) myInSlot.pop();
```

The *pop()* method blocks until new data is available in the InSlot. For non-blocking operation,

you can check if new data is available before calling *pop()* by calling the *empty()* method:

C++
```
if (!myInSlot.empty())
{
  bool value = myInSlot.pop();
  // process the new data value
  …
}
```
Java
```
if (!myInSlot.empty())
{
  Boolean value = (Boolean) myInSlot.pop();
  // process the new data value
  …
}
```

When calling the *pop()* method, the data value is removed from the InSlot, i.e. each call to *pop()* returns a new data value. To get the newest data value without actually removing it from the InSlot, you can call the *top()* method. *top()* blocks until a new data value is available. It then returns that data value from the InSlot without removing it, i.e. the next call to either *top()* or *pop()* returns the same data value:

C++
```
bool value = myInSlot.top();
```
Java
```
Boolean value = (Boolean) myInSlot.top();
```

The examples for the *pop()* and *top()* methods shown so far just demonstrate how to get the data values. Of course it is also possible to retrieve the corresponding timestamps. When using C++, *pop()* and *top()* actually return InstantIO *Data* objects that are automatically casted to the data values in the examples above. To get the timestamps, you simply have to use these InstantIO *Data* objects:

C++
```
Data<bool> data = myInSlot.pop();
unsigned long timestamp = data.getTimeStamp();
bool value = data.getValue();
```

In Java, there is no such concept of automatic type transformations. For that reason, you have to use use two special methods instead, *popData()* and *topData()*:

Java
```
Data data = myInSlot.popData();
Long timestamp = data.getTimeStamp()
Boolean value = (Boolean) data.getValue();
```

InSlots by default are not buffered, i.e. when a new data value is received from an OutSlot before reading the previous value, the previous value gets lost. That behaviour in many cases is desired, i.e. when you receive position values from a tracking system, you are usually only interested in the most recent position. But in other cases, you need to get all data values, i.e. when receiving button press events. For this reason, a special descendant of the InSlot class exists, the

BufferedInSlot, that buffers data values. When you create such a BufferedInSlot, you have to specify the buffer size, i.e. the number of data values that can be buffered:

C++

```
BufferedInSlot<bool> myBufferedInSlot(10);
```

Java

```
BufferedInSlot myBufferedInSlot =
  new BufferedInSlot(Boolean.class, 10);
```

In both examples, a BufferedInSlot with a buffer size of 10 data values is created. When data values at the InSlot, they are added to the buffer. When there is no free space in the buffer, the oldest data value in the buffer gets overwritten. When you call the *pop()* method, the oldest data value is removed from the buffer and returned. Besides the fact that BufferedInSlots buffer data values, they behave exactly like normal InSlots, unbuffered InSlots.

Just like OutSlots, InSlots have a *isConnected()* method that allows to check if any OutSlot is connected to the InSlot, even though that method is rarely used:

C++

```
if (myInSlot.isConnected())
{
  …
}
```

Java

```
if (myInSlot.isConnected())
{
  …
}
```

Both OutSlots and InSlots allow to add so-called listeners that get notified when the status of the slot changes. You have to inherit your own class from the abstract listener interface and implement the methods of the interface. Then, you have to create an instance of your listener class and add it to the slot.

For OutSlots, you have to implement the *Listener* interface declared inside the OutSlot class. That interface has two methods you have to implement, *startOutSlot(…)* and *stopOutSlot(…)*. *startOutSlot(…)* gets called when the first InSlots connects to the OutSlot, i.e. when you have to start writing data values into the OutSlot. *stopOutSlot(…)* gets called when the last InSlot disconnects from the OutSlot, i.e. when have to stop writing data values into the OutSlot. To add your listener to the OutSlot, call its *addListener(…)* method. To remove your listener from the OutSlot, call its *removeListener(…)* method.

C++

```
class MyOutSlotListener: public BasicOutSlot::Listener
{
  virtual ~MyOutSlotListener() {};
  virtual void startOutSlot(BasicOutSlot &outSlot)
  {
    // Gets called when the first InSlot connects to
    // the OutSlot
```

```
  };
  virtual void stopOutSlot(BasicOutSlot &outSlot)
  {
    // Gets called when the last InSlot disconnects from
    // the OutSlot
  };
};
…
MyOutSlotListener myOutSlotListener;
OutSlot<bool> myOutSlot;
myOutSlot.addListener(myOutSlotListener);
…
myOutSlot.removeListener(myOutSlotListener);
```

Java

```
class MyOutSlotListener implements OutSlot.Listener
{
  public void startOutSlot(OutSlot outSlot)
  {
    // Gets called when the first InSlot connects to
    // the OutSlot
  };
  public void stopOutSlot(OutSlot outSlot)
  {
    // Gets called when the last InSlot disconnects from
    // the OutSlot
  };
};
…
MyOutSlotListener myOutSlotListener = new MyOutSlotListener();
OutSlot myOutSlot = new OutSlot(Boolean.class);
myOutSlot.addListener(myOutSlotListener);
…
myOutSlot.removeListener(myOutSlotListener);
```

For InSlots, you have to implement the *Listener* interface declared inside the InSlot class. That interface has three methods you have to implement, *startInSlot(…)*, *stopInSlot(…)* and *newData(…)*. *startInSlot(…)* gets called when the first OutSlots connects to the InSlot, i.e. when you have to start reading data values from the InSlot. *stopInSlot(…)* gets called when the last OutSlot disconnects from the InSlot, i.e. when have to stop reading data values from the InSlot. *newData(…)* gets called when there is a new data value available for receival from the InSlot. To add your listener to the InSlot, call its *addListener(…)* method. To remove your listener from the InSlot, call its *removeListener(…)* method.

C++

```
class MyInSlotListener: public BasicInSlot::Listener
{
  virtual ~MyInSlotListener() {};
  virtual void startInSlot(BasicInSlot &InSlot)
  {
    // Gets called when the first OutSlot connects to
    // the InSlot
  };
  virtual void stopInSlot(BasicInSlot &inSlot)
  {
    // Gets called when the last OutSlot disconnects from
    // the InSlot
  };
  virtual void newData(BasicInSlot &inSlot)
  {
    // Gets called when there is new data available for
    // receival from the InSlot
  }
};
…
MyInSlotListener myInSlotListener;
InSlot<bool> myInSlot;
myInSlot.addListener(myInSlotListener);
…
myInSlot.removeListener(myInSlotListener);
```

Java

```
class MyInSlotListener implements InSlot.Listener
{
  public void startInSlot(InSlot inSlot)
  {
    // Gets called when the first OutSlot connects to
    // the InSlot
  };
  public void stopInSlot(InSlot inSlot)
  {
    // Gets called when the last OutSlot disconnects from
    // the InSlot
  };
  public void newData(InSlot inSlot)
  {
    // Gets called when there is new data available for
    // receival from the InSlot
  }
};
…
MyInSlotListener myInSlotListener = new MyInSlotListener();
InSlot myInSlot = new InSlot(Boolean.class);
myInSlot.addListener(myInSlotListener);
…
myInSlot.removeListener(myInSlotListener);
```

Keep in mind that the methods of the Listener objects usually get called from another thread or even multiple other threads – so make sure that your implementations are thread-safe!

Namespaces and Routes

Namespaces and Routes are components of the mechanism that automatically connects OutSlots and InSlots. As mentioned before, applications are able to connect OutSlots and InSlots manually. But the problem is that usually software components that need to exchange data do not know about the slots of other software components. Therefore, we need a more abstract way of connecting OutSlots and InSlots.

This level of abstraction is performed by Namespaces and Routes. An application creates Namespace objects. Then, it adds all its OutSlots and InSlots to these Namespaces. When doing this, it has to give a name to each slot. This name neither has to be unique, nor it has to follow any conventions, it simply is a label. For example, in the case of a tracker that tracks the position of the user's head, the application would add an OutSlot that provides position and orientation data to the Namespace under the name "Tracker 1/Sensor 1", and an InSlot that receives position and orientation data under the name "Head position". Case does not matter for labels.

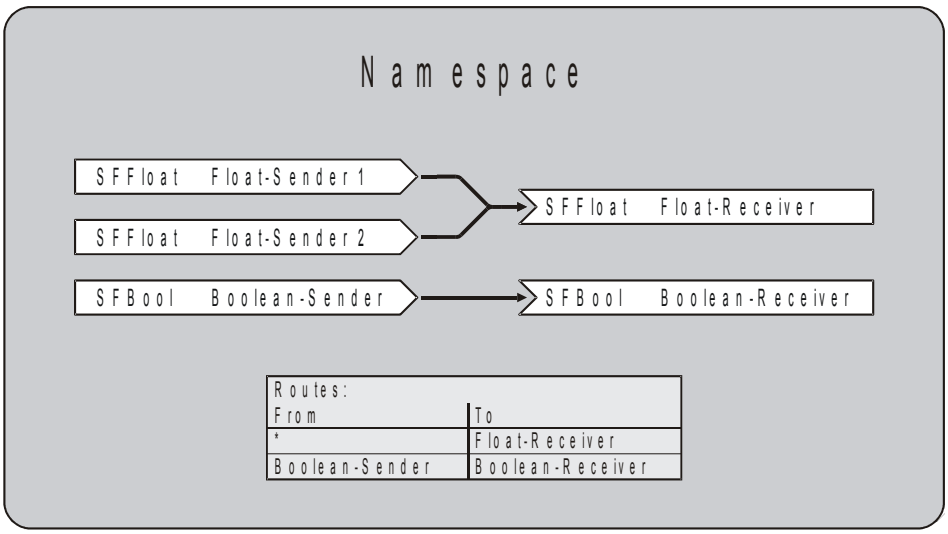

Figure 4: Namespaces and Routes are used to connect OutSlots and InSlots on a higher level of abstraction

The following example demonstrates how to create Namespace objects and how to add slots to them using the *addOutSlot(…)* and *addInSlot(…)* methods of the Namespace:

C++

```
Namespace ns;

ns.addOutSlot("Tracker 1/Sensor 1", &myOutSlot);
ns.addInSlot("Head position", &myInSlot);
```

Java

```
Namespace ns = new Namespace();

ns.addOutSlot("Tracker 1/Sensor 1", &myOutSlot);
ns.addInSlot("Head position", &myInSlot);
```

It is perfectly ok to add slots more than once to a Namespace using the same or a different label.

To remove slots from a Namespace, use the *removeOutSlot(…)* and *removeInSlot(…)* methods. When calling these methods, you have to specify the labels you gave the slots when you added them to the Namespace:

C++

```
ns.removeOutSlot("Tracker 1/Sensor 1", &myOutSlot);
ns.removeInSlot("Head position", &myInSlot);
```

Java

```
ns.removeOutSlot("Tracker 1/Sensor 1", &myOutSlot);
ns.removeInSlot("Head position", &myInSlot);
```

When you add a slot n times under the same label to the Namespace, you also have to remove the slot n times before it actually gets removed from the Namespace.

To connect OutSlots and InSlots, the application now adds Routes to the Namespace that map OutSlot names to InSlot names. For example, to connect the OutSlot of the tracking component with the InSlot of the rendering component in the example given above, the application would

add a Route to the Namespace that maps the name “Tracker 1/Sensor 1” to the name “Head position” (again, case does not matter):

C++ / Java

```
ns.addRoute("Tracker 1/Sensor 1", "Head position");
ns.enable();
```

It is not possible to add a Route more than once to a Namespace – when you try to add a Route that already exists in the Namespace, the call is silently ignored.

You can also use wildcards like “?”, “*” and square brackets “[]” in Routes. “?” stands for exactly one arbitrary character. “*” stands for an arbitrary number of arbitrary characters. In square brackets, you can specify the characters that are allowed at that position. “[1-9]” means that the characters from “1” to “9” are allowed at this position. “[aeiou]” means that the characters “a”, “e”, “i”, “o”, and “u” are allowed at this position. To use the wildcard characters without their special meaning, escape them using the backslash character “\”.

Routes automatically connect OutSlots and InSlots in the Namespace when:

- OutSlot and InSlot have the same type.
- The label of the OutSlot in the Namespace matches the first label of the Route (the “from” part of the Route).
- The label of the InSlot in the Namespace matches the second label of the Route (the “to” part of the Route).

You do not have to create Routes for OutSlots and InSlots that have the same label – slots with the same label connect automatically when their types match.

Before a Namespace actually connects OutSlots and InSlots as specified by the Routes, it has to be enabled by calling the *enable()* method. The order of method calls is not important – you can also enable the Namespace directly after creating it and before adding the slots and the Routes. You just have to keep in mind that the Namespace only connects slots when it is enabled.

To remove a Route from a Namespace, call the *removeRoute()* method. This does not only remove the Route, but also disconnects all slots affected by this Route. When you try to remove a Route that does not exist, the call is silently ignored. To disconnect all slots in the Namespace, you can simply call the *disable()* method of the Namespace:

C++ / Java

```
ns.removeRoute("Tracker 1/Sensor 1", "Head position");
ns.disable();
```

The mechanism of namespaces and routes allows software components to connect OutSlots and InSlots without actually knowing the instances of the slots. But a problem still remains: How can different software components add their slots to the same Namespace instance? Somehow they have to agree on a single Namespace instance. The solution is a special “Root” Namespace. This namespace is a singleton provided by the device management system and is the central place where software components exchange data. You get the single instance of the Root Namespace by calling the static *the()* method of the *Root* object:

C++

```
Root &root = Root::the();
```

Java

```
Root root = Root.the();
```

You can use the *Root* object the same way as a *Namespace* object – the *Root* class actually is a descendant of the *Namespace* class. The only difference is that you can create any number of instances of *Namespace* objects, but there is only one instance of the *Root* object per process. For this reason, the single *Root* object is the central place where different software components exchange data.

Subnamespaces and External Routes

Namespaces and Routes are an important mechanism to interconnect OutSlots and InSlots of different software components. But real applications have to deal with a large number of slots, and putting them all into one single Namespace results in a very unclear application structure. The solution is the concept of Subnamespaces. You cannot only add slots to Namespaces, you can also add other Namespaces as Subnamespaces, allowing to build a hierarchical tree of Namespaces. In this tree, "External Routes" can be used to export slots from Subnamespaces to their parent Namespaces. This allows to structure the system into smaller units. The application developer is able to hide implementation details into Subnamespaces.

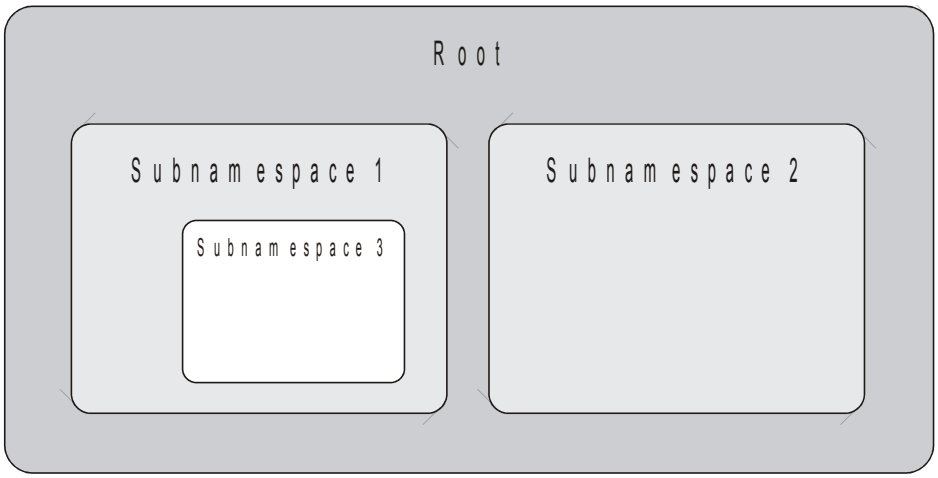

Figure 5: Namespaces can be nested. A single Root namespaces is provided by the system

To add a Namespace to another Namespace, use the *addNamespace(…)* method. Just like slots, you have to label subnamespaces when you add them to Namespaces, but unlike slots, this label has to be unique, and you can put a Namespace only once into another Namespace:

C++

```
Namespace ns;
std::auto_ptr<Namespace> subns = new Namespace;

std::string label = ns.addNamespace("Subnamespace", subns);
```

Java

```
Namespace ns = new Namespace();
Namespace subns = new Namespace();

ns.addNamespace("Subnamespace", subns);
```

The *addNamespace(…)* method takes the label and a pointer to the subnamespace as parameters. When the label already exists in the parent Namespace, this method automatically makes the label unique by adding a number to the label. It returns the unique name the subnamespace actually got in the parent Namespace. The parent Namespace takes full control over the subnamespace, i.e. it automatically deletes all subnamespaces when it gets deleted, and it enables resp. disables all subnamespaces when it gets enabled or disabled.

There exists also an overloaded convenience version of *addNamespace(…)* that does not take a label. In this case, the method automatically creates a unique label, usually "Namespace" followed by a number:

C++

```
Namespace ns;
std::auto_ptr<Namespace> subns = new Namespace;

std::string label = ns.addNamespace(subns);
```

Java

```
Namespace ns = new Namespace();
Namespace subns = new Namespace();

ns.addNamespace(subns);
```

To remove a subnamespace from a Namespace, use the *removeNamespace(…)* method. This method takes one parameter, the unique label returned by the previous call to *addNamespace(…)*:

C++

```
std::auto_ptr<Namespace> subns = ns.removeNamespace(label);
```

Java

```
Namespace subns = ns.removeNamespace(label);
```

removeNamespace(…) returns the subnamespace it removed from the parent namespace. You are free to add this Namespace to another Namespace, or to use it any way you need to. When you do not need the Namespace anymore, you have to delete it.

Putting a Namespace into another Namespaces of course only makes sense when you export slots from the subnamespace into the parent Namespace. To do that, you have to create External Routes. Like normal Routes, External Routes consist of two labels that specify the internal name of the slot, i.e. its name in the subnamespace, and the external name of the slot, i.e. its name in the parent Namespace. To create an External Route, call the *addExternalRoute(..)* method of the subnamespace:

C++ / Java

```
subns.addExternalRoute("Tracker 1/Sensor 1", "Head position");
```

The *addExternalRoute(…)* takes two labels as parameters, the internal label, and the external label. In the example above, all OutSlots and InSlots with the label "Tracker 1/Sensor 1" in the subnamespace get exported under the label "Head position" into the parent Namespace.

For the internal label, you can use the wildcards “*”, “?” and “[]”. “?” stands for exactly one arbitrary character. “*” stands for an arbitrary number of arbitrary characters. In square brackets, you can specify the characters that are allowed at that position. “[1-9]” means that the characters from “1” to “9” are allowed at this position. “[aeiou]” means that the characters “a”, “e”, “i”, “o”, and “u” are allowed at this position. To use the wildcard characters without their special meaning, escape them using the backslash character “\”.

For the external label, you cannot use wildcards, instead you can two placeholders:

- “{SlotLabel}” gets replaced by the original name of the slot in the subnamespace.
- “{NamespaceLabel}” gets replaced by the name of the subnamespace.

So for example, to export all slots under their original name into the parent namespace, you just have to add one single External Route:

C++ / Java
```
subns.addExternalRoute("*", "{SlotLabel}");
```

To export all slots under their original name, but prefixed by the name of the subnamespace, add the following External Route:

C++ / Java
```
subns.addExternalRoute("*", "{NamespaceLabel}/{SlotLabel}");
```

Keep in mind that the subnamespace has to be enabled, otherwise External Routes do not become active.

To remove an External Route, simply call the *removeExternalRoute(…)* method:

C++ / Java
```
subns.removeExternalRoute("Tracker 1/Sensor 1", "Head position");
```

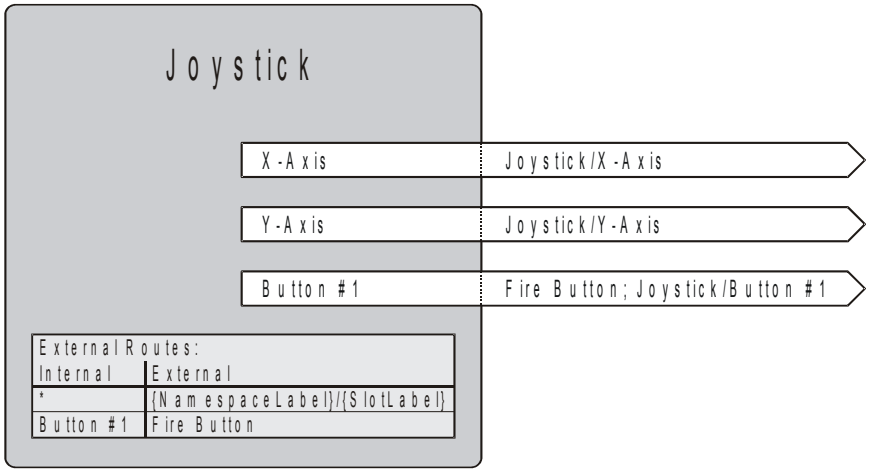

Figure 6: External Routes export slots into the parent namespace

Data Types and Smart Pointer

As mentioned before, the InstantIO system is able to handle any data type. When creating an

OutSlot or an InSlot, the application is free to specify any data type that can be transferred by that slot. Nevertheless, the InstantIO system provides some standard data types for position values, rotations, matrices, video frames etc. These data types do not have functionality, they are not meant to be used inside the application to represent vectors and matrices. Instead, they are only containers that allow to transfer data values in a standardized way. This ensures interoperability between different software components, i.e. software components that stick to these standard data types ensure that they will smoothly collaborate with software components written by other persons or for other projects.

The following table shows all data types provided by the system. For more information about how to use these data types, see the respective API documentation.

C++	Java	Description
InstantIO::Color	org.instantreality.InstantIO.Color	A vector of three float components storing the red, green and blue color components as values between 0 (no intensity) and 1 (full intensity)
InstantIO::ColorRGBA	org.instantreality.InstantIO.ColorRGBA	A vector of four float components storing the red, green, blue and alpha color components as values between 0 (no intensity) and 1 (full intensity)
InstantIO::Image	org.instantreality.InstantIO.Image	A bitmap for storing video frames
InstantIO::Matrix3d	org.instantreality.InstantIO.Matrix3d	A 3x3 matrix of double components
InstantIO::Matrix3f	org.instantreality.InstantIO.Matrix3f	A 3x3 matrix of float components
InstantIO::Matrix4d	org.instantreality.InstantIO.Matrix4d	A 4x4 matrix of double components
InstantIO::Matrix4f	org.instantreality.InstantIO.Matrix4f	A 4x4 matrix of float components
InstantIO::Rotation	org.instantreality.InstantIO.Rotation	A rotation stored in a quaternion
InstantIO::Time	–	An unsigned long value counting milliseconds
InstantIO::Vec2d	org.instantreality.InstantIO.Vec2d	A vector of two double components (x and y)
InstantIO::Vec2f	org.instantreality.InstantIO.Vec2f	A vector of two float components (x and y)
InstantIO::Vec3d	org.instantreality.InstantIO.Vec3d	A vector of three double components (x, y and z)
InstantIO::Vec3f	org.instantreality.InstantIO.Vec3f	A vector of three float components (x, y and z)
InstantIO::Vec4d	org.instantreality.InstantIO.Vec4d	A vector of four double components (x, y, z and w)
InstantIO::Vec4f	org.instantreality.InstantIO.Vec4f	A vector of four float components (x, y, z and w)

Outslots copy data values written into them to all inslots that are connected to the outslot. This is no problem for data values like vectors or rotations, but it would be extremely inefficient for data types like video frames that require to copy a large amount of memory. For this reason, on C++, thread-safe smart pointers are used for such kinds of data types. Instead of copying the data values, the outslots just copy the pointer to the data value. The memory consumed by the data value is automatically released when the last pointer to the data is destroyed. See the API documentation for InstantIO::SmartPtr for more information about how to use smart pointers. On Java, all data values are actually pointers to data objects, and copying data values into InSlots just

means copying pointers. For this reason, there is no need for a special smart pointer concept in the InstantIO Java language binding.

High-level Interface

In contrast to the low-level interface that deals with the communication between different software components, the high-level interface allows to build data flow graphs consisting of nodes that are state machines, receive data values on incoming edges, change their state depending on the incoming data, and send data values on their outgoing edges.

Nodes

Nodes are the core component of the data flow graph. There are four types of nodes:

1. Nodes that produce data. These are usually device drivers of input devices, nodes that replay data streams stored on a hard disk, timers etc.
2. Nodes that transform data. Examples are nodes that transform coordinate systems, or video tracking systems that take video frames and transform them to position and orientation values.
3. Nodes that consume data. These are usually device drivers of output devices, nodes that store data streams on a hard disk, etc.
4. Nodes that do not deal with data in any way. This sounds strange at a first glance, but our system currently has already three nodes of this kind: The “Network” node that makes the system network transparent, the “Web” node that provides a user interface to the framework, and the “Inline” node that allows integrate subgraphs stored in configuration files.

To create a new node, the developer has to derive a new class from the abstract node base class. The base node class itself is a descendant of the namespace class, i.e. each node inherits the namespace interface and can be handled exactly the same way.

Then, the programmer has to create the outSlots and inSlots used to communicate with other nodes. This is usually done in the constructor. Sometimes the number of slots varies depending on the device that is actually handled by the node, e.g. when operating a joystick with two axes and one button, the joystick device driver node creates two float outslots and a boolean outslot. All slots are given a self-explaining name and are added to the node (remember that nodes are derived from namespaces). For example, the two float outslots of the joystick node get the labels “Joystick 1/X-Axis” and “Joystick 1/Y-Axis”, and the boolean outslot gets the label “Joystick 1/Button #1”. Nodes usually export all slots contained in them to the parent namespace.

An important design decision that has to be made by the developer of a new node is whether the node uses its own thread or not. Device driver nodes for input devices usually always need to have an own thread, because they have to listen to the port the device is connected to. All other types of nodes do not necessarily need to have their own thread. For example, filter nodes that transform position values into another coordinate system usually directly calculate the new values and write them to the outslots when they are notified by the inslot that new data is available for reading. This means that they are in fact driven by the threads of the device driver nodes that provided the incoming data values. On the other hand, filter nodes that have to perform lengthy calculations, like video trackers, usually get their own threads.

When the application builds the data flow graph, it creates the nodes, adds them to a namespace, and connects their outslots and inslots using routes.