VeniceHub

Manual

April 9, 2015

Contents

1 Introduction **3**
1.1 Special characters in this manual 3

2 Installation **3**

3 Usage **3**
3.1 Start and quit VeniceHub 3
3.2 Using options 3
3.3 Mode of operation 4
3.4 Logging data to disk 4
3.5 Replaying data from file 4
3.6 Receiving data from InstantIO 5
3.7 Sending data to InstantIO 5
3.8 RSB receiving and sending 5
3.9 VenicePort (VP) - Connection to TCP socket 6
3.10 More options 6

4 XIO lines **7**
4.1 General Syntax 7
4.2 Parsing of special types 7

5 Fileformat for VeniceHub log files **8**

6 Supported data types **8**

7 XIO codes **8**

8 Slot file **9**

9 Format for VP strings **10**

10 Protobuf Classes **11**
10.1 How to create a custom RSB protobuf class file 11

11 Class matching between RSB and InstantIO **12**
11.1 Example Float 13
11.2 Example for multifield classes 13
11.3 Example singlefield protobuf class with more than one value 14
11.4 Example for a multifield protobuf class with more than one value per field 14

12 Runtime Commands **15**
12.1 config 15
12.2 h 15
12.3 msg 15
12.4 offset [{x}] 15
12.5 p 15
12.6 reset 15
12.7 savelag 15
12.8 seek {timestamp} 15
12.9 skip {time} [ms | s | m | h] 15
12.10 time 15
12.11 xio 15

13 Troubleshooting **16**
13.1 Multicast 16
13.2 Quit 16

14 How to setup some Venice scenarios **16**
14.1 VeniceHub and Kinect 16
14.2 VeniceHub and ELANMod 16
14.3 Translating IIO $\leftrightarrow$ RSB 17
14.4 Multiple data sources, multiple data loggers 18

15 Tables and Figures **19**

1 Introduction

VeniceHub is a platform independent software that connects different sources of sensor data. It receives and sends sensor data via network, can log data to a log file on disk and replay it in the same timescale it was recorded. When replaying logged sensor data, VeniceHub can receive remote commands to play, pause and seek specific time positions.
The supported network protocols are InstantIO and RSB and for (de)compression of logged data GZIP is used.

1.1 Special characters in this manual

In this manual the square brackets "`[`" and "`]`" are used to mark optional parameters, optional arguments and optional content.
Curly brackets "`{`" and "`}`" are used to mark placeholders.

2 Installation

Download the release version and copy the content into a directory of your choice.
To run VeniceHub, java (1.7 or higher) needs to be installed.
To use InstantReality with VeniceHub, get `instantreality.jar` from www.instantreality.org and copy it into the subfolder VeniceHub_lib.
To use Robotic Service Bus (RSB) (including Google Protocol Buffers) two java jars are necessary:
`rsb-0.10.0.jar` (or higher) from
http://docs.cor-lab.org/rsb-manual/trunk/html/install-binary.html#pre-compiled-jar-download
and
`protobuf-java-2.4.1.jar` (or higher) from
http://mvnrepository.com/artifact/com.google.protobuf/protobuf-java

3 Usage

3.1 Start and quit VeniceHub

The command to start VeniceHub is:

```
java -jar VeniceHub.jar [{options}]
```

On some operation systems (e.g. Linux) there may be a problem with multicast connection, see section 13.1 (page 16) for more information.
Without options, VeniceHub will operate in default mode with default values. The default mode is Logging data from InstantIO to a file named `log.xio.gz`[1].
To quit VeniceHub, enter `q`.

3.2 Using options

Options can be given in two ways: By command line arguments or by file. For example the option to give a filename is `--file {filename}`. For some options exists a short form. For example the short form for `--file` is `-f`. To see all possible options, use the `--help` option (short `-h`).

To give the option by file, name the file with an @, for example:

```
java -jar VeniceHub.jar @myOptions.txt
```

Then the file `myOptions.txt` can contain options like this:

[1]If this file already exists, the name will be changed, see section 3.4

```
--input
Disk

--output
IIO

--file
mylogfile.log
```

It is possible to mix both methods. For example:

```
java -jar VeniceHub.jar @myOptions.txt -s sensor.xml
```

It is not allowed to specify an option more than once.

3.3 Mode of operation

VeniceHub knows different modes of operation, depending on input source and output target. The default mode is logging data from InstantIO to Disk. The mode is set by using the options `--input` and `--output`, or short `-i` and `-o`. The parameters for those options are shown in table 1.

input	output	remark
Disk	IIO	replaying from disk to InstantIO
Disk	RSB	replaying from disk to RSB
IIO	Disk	logging from InstantIO to disk (default mode)
IIO	RSB	receiving from InstantIO, converting and sending to RSB
RSB	Disk	logging from RSB to disk
RSB	IIO	receiving from RSB, converting and sending to InstantIO
VP	IIO	receiving from TCP port and sending to InstantIO
VP	RSB	receiving from TCP port and sending to RSB
VP	Disk	receiving from TCP port and logging to disk

Table 1: Modes of operation

3.4 Logging data to disk

By default VeniceHub will log incoming data from InstantIO to disk. Starting VeniceHub in default mode does not require any options:

```
java -jar VeniceHub.jar
```

The default name of the log file is `log.xio.gz`. With the option `--file {filename}` the filename can be changed. If a file with this name already exists, an index will be added to the filename. If this new filename also exists, the index will be increased until there is no existing file with this name.
The data from network will be converted into a string of the format of a XIO line, see section 4 for more information about XIO lines used by VeniceHub.

The data will be compressed using GZIP (so it is recommended to use the suffix `.gz` for the file name). With the `--writeRaw` option the data will be written without compression.

By default, VeniceHub will log everything received over InstantIO. To select specific sensor fields from InstantIO namespace or RSB scope, a sensor definition file is required, see section 8.

3.5 Replaying data from file

When replaying from disk, VeniceHub will parse the content of the file and send the data into network.
For example

```
java -jar VeniceHub.jar -i Disk -o IIO -f mylog.xio.gz
```

will replay the data in `mylog.xio.gz` into InstantIO network.
The file can be compressed using GZIP and has to contain XIO lines, see section 4. The name of the file has to be defined with the `--file` option (or `-f`), otherwise the default will be used (`log.xio.gz`).
The sensorname values of the XIO lines will be used as scopes (to create namespace and slotlabel), so for example, for the XIO line to be replayed via InstantIO

```
<sfint32 value="1.234" timestamp="1390123456789" sensorname="sim/car/velocity"/>
```

VeniceHub will create a slot labeled with `velocity` and the namespace `sim/car` (assuming that no prefix for namespaces was given).

3.6 Receiving data from InstantIO

To receive data from InstantIO network, multicast needs to be enabled and the multicast options `--mcadress`, `--mcport` and `--mcttl` are relevant. The default settings are:

```
--mcadress 224.21.12.68
--mcport 4711
--mcttl 0
```

VeniceHub will create a network node and a listener to each slot it detects. Data is then received from every slot. Be sure to increase the TTL (Time-To-Live), if a connection with one ore more routers is used (each router will reduce the TTL for a datagram and if it reaches 0 it will be discarded).

3.7 Sending data to InstantIO

Sending data to InstantIO is possible in two ways.

The first way is using dynamic slot creation (default). For each data item a corresponding slot is created 'on the fly', if a slot with that name is not already existing. The name of the slot will be derived from the sensorname of the data item. InstantIO needs some time to initialize new slots and will silently ignore data being send to those slots while initializing.

The second way is to predefine slots with a sensor file (see section 8 for more information about sensor files). A data item will only be send to InstantIO, if its sensorname matches a given namespace and slot name.

The data item is then converted into a instantreality data object of the corresponding type and send to the matching slot. Not all data types are supported by the Java implementation of InstantIO, see section 6 for more information about supported data types.

3.8 RSB receiving and sending

To use RSB with other data types than Strings, protobuf definitions of those types are necessary. The folder with the protobuf class files is given by the option `--protobuf {path}`. The default setting for protobuf classes is `--protobuf ./protobuf`, so if this option is not given by the command line arguments, VeniceHub will search the current directory for a folder named protobuf and will use this, if found. See section 10 for more information on how to use protobuf classes with RSB.
The default scope for sending can be given with the option `--rsbDefInfScope`.

It is possible to use RSB without protobuf classes, but then only string objects are supported. Two options can help here: With the option `--rsbStringsAreXIO` VeniceHub will parse received String objects as XIO lines. With the option `--rsbtoXIO` VeniceHub will send complete XIO lines as Strings via RSB. But this is only a workaround for the case that protobuf class files are not available.

It is also possible to use predefined scopes, by giving a sensor file (see section 8 for more information about sensor files). A data item will only be send or received via RSB if it's sensorname matches a given scope and informer name.

The easiest way is to use default values for all RSB options.

Example for logging from RSB to Disk:

```
java -jar VeniceHub -i RSB -o Disk
```

VeniceHub will then assume that the current directory contains a protobuf folder and a file named xiocodes_RSB.xml (both are already included in the release version).

Example for translating from IIO to RSB:

```
java -jar VeniceHub -i IIO -o RSB
```

VeniceHub will then assume that the current directory contains a protobuf folder, a file named xiocodes_RSB.xml and a file named match.xml (all three are already included in the release version).

3.9 VenicePort (VP) - Connection to TCP socket

In VP mode, VeniceHub reads from a TCP socket. VP mode is only accepted as input. Needed options are:

```
--vpport {port number}
--vpfile {name of XML file with slot definitions}
```

With `--vpport` the TCP port is given. The reader thread of VeniceHub will wait until a client connects to this port and then starts to parse the incoming strings. For the parsing a XML file is needed, which defines the structure of the string. This XML file is given by `--vpfile`, or short `-v`. see section 9 for more information how the format of the string is defined.

3.10 More options

To see all options, use `-help` or `-h`. In table 2 some option are listed.

Option	Explanation
`--headerlines {n}`	number of headerlines to ignore (default 2)
`--rpcServerAdress {adress}`	adress for RPC (default localhost)
`--rpcServerPort {port}`	port for RPC (default 4243)
`--offset {ms}`	Setting offset for synchronization of replay with ELAN
`--slotfile {filename}`	Giving a XML file with predefined slots
`--sendInitValue`	activate sending of initialization values for predefined IIO out-slots
`--mcadress {adress}`	adress for multicast
`--mcport {port}`	port for multicast
`--mcttl {n}`	time to live for multicast (default 2)
`--parser {n}`	which parser to use (DOM or REGEX) (default REGEX)
`--rsbDefInfScope {scope}`	sets default scope for RSB (default /)
`--classMatcher {filename}`	Giving a XML file with class matching definitions for RSB
`--protobuf {path}`	path for protobuf files for RSB types
`--queueCapacity {events}`	set queue size for event queue (default 10000)
`--bufferCapacity {lines}`	set disk reader buffer capacity (default 1000000)
`--writeRaw`	writes data to file without compression

Table 2: Some options

4 XIO lines

4.1 General Syntax

To log data, VeniceHub converts the data from each sensor into a XIO line and writes it into the log file. When replaying data from a log file, VeniceHub trys to parse the content of the file as XIO lines.
XIO lines have to be in the following format:

```
<{type} value="..." timestamp="..." sensorname="..."/>
```

The `{type}` above will be replaced by the data type of the sensor, see section 6 for more information about supported data types. If the data type of the sensor is not supported, the default type `sfstring` will be used.
If a change to the default mapping between data types and XIO codes is needed, see section 7 for more information.
The `timestamp` will be the current system time in milliseconds. The `sensorname` will be the complete scope of the sensor, consting of a slotlabel and an optional namespace, separated by slahes (in the format namespace/slotlabel).
An example for a XIO line:

```
<sffloat value="42.15" timestamp="1391234567890" sensorname="sim/car/speed"/>
```

The value may not contain the special characters for new line or carriage return. The sensorname (=scope) consists of the namespace `sim/car` and the slotlabel `speed`.

Additional to the above mentioned syntax, there is an older format, also supported, but deprecated:

```
<irio:{type} value="..." timestamp="..." sensorname="..."></irio:{type}>
```

Note: The regular expression matching parser (RegEx) that VeniceHub uses by default, may get confused by minor changes of this syntax, although those changes my be XML-conform.

4.2 Parsing of special types

While simple types (sfstring, sffloat, sfint32, sfdouble, sfboolean) are parsed in an obvious way, the parsing of more complex types is explained in this subsection.

`sfec2f, sfvec3f, sfrotation`: Those one-dimensional multi-element types are converted into a string element by element. Those strings are concatenated with one space as a delimiter. For example a `sfvec3f` with the value of `0.1, 0.2, 0.3` becomes:

```
<sfvec3f value="0.1 0.2 0.3" timestamp="..." sensorname="..."/>
```

Multi fields: Multi field types are first disassembled field by field. Each field is converted into a string. Then all strings are concatenated with exactly one comma and one space as a delimiter. Additionally an opening square bracket is put in front of the resulting string and a closing square bracket is put at the end of the resulting string. For example a mfint32 consisting of four integers with the values `1, 3, 5, 7` is converted to:

```
<mfint32 value="[1, 3, 5, 7]" timestamp="..." sensorname="..."/>
```

A multi field version of a multi-element type will first be disassembled field by field and then each field will be converted into a string according the appropriate rules. For example, a mfvec2f consisting of three vectors with the values `1.1, 1.2` and `2.1, 2.2` and `3.1, 3.2` will be converted to:

```
<mfint32 value="[1.1 1.2, 2.1 2.2, 3.1 3.2]" timestamp="..." sensorname="..."/>
```

Please note that the fields are delimited by one comma and one space, while the elements *inside* a field are delimited by one space.

5 Fileformat for VeniceHub log files

The file format:

```
...some header lines...
<venice>
...XIO lines...
</venice>
```

Usually the filename ending is `.xio`. The file may be compressed with gzip, then the filename ending is usually `.xio.gz`.

The XIO lines in the file must be terminated by an end-of-line character. They can not include an end-of-line character (for example an EOF in the value of an sfstring will break the parsing).

The number of header lines to be ignored are by default 2 and can be changed with the option `--headerlines`.

By default VeniceHub compresses data written to disk with gzip. To write uncompressed data, use the option `--writeRaw`.

6 Supported data types

Data types supported by VeniceHub for InstantIO and disk are shown in table 3. Only a few non-instantreality types are supported by the Java implementation of InstantIO and not all of them are supported as multi fields. For now not all of the instantreality types are supported by VeniceHub, only those, we used in preceding experiments (mostly Vec2f, Vec3f and Rotation).

Type	single field	multi field	remark
Boolean	sfbool	-	
String	sfstring	mfstring	should not contain new lines, nor carriage return
Integer	sfint32	-	
Float	sffloat	-	
Double	sfdouble	-	
Vec2f	sfvec2f	mfvec2f	
Vec3f	sfvec3f	mfvec3f	
Rotation	sfrotation	mfrotation	

Table 3: Supported datatypes

The Java implementation of InstantIO will silently ignore unsupported data types.

For RSB all protobuf classes in the protobuf folder are supported.

The mapping what XIO code belongs to what type can be changed with the option `--xiocodes {filename}`. For InstantIO there are default values for all supported types. But for RSB there is only `sfstring` by default, so naming this mapping file is mandatory if more datatypes than strings are needed. See section 7 for more information.

7 XIO codes

For InstantIO by default the XIO code `sfstring` is used for string objects, `sffloat` for floating point values, and so on. If this mapping needs to by changed, or if RSB protobuf classes are used, the option `--xiocodes {filename}` is necessary. With this option a file is named, that contains a XML structure with definitions on how to map types to XIO codes. A file that would map some supported types of InstantIO to the default XIO codes would look like this:

```
<?xml version="1.0"?>
<codes>
<def class="java.lang.String" code="sfstring"/>
<def class="[Ljava.lang.String;" code="mfstring"/>
<def class="java.lang.Float" code="sffloat"/>
```

```
<def class="[Ljava.lang.Float;" code="mffloat"/>
<def class="org.instantreality.InstantIO.Vec3f" code="sfvec3f"/>
<def class="[Lorg.instantreality.InstantIO.Vec3f;" code="mfvec3f"/>
...
</codes>
```

For RSB protobuf classes it can look like this:

```
<?xml version="1.0"?>
<codes clear="true">
<def class="protobuf.Int32Protos$Int32" code="sfint32"/>
<def class="protobuf.BoolProtos$Bool" code="sfbool"/>
<def class="protobuf.FloatProtos$Float" code="sffloat"/>
<def class="protobuf.MFFloatProtos$MFFloat" code="mffloat"/>
<def class="protobuf.Vec2fProtos$Vec2f" code="sfvec2f"/>
<def class="protobuf.MFVec2fProtos$MFVec2f" code="mfvec2f"/>
<def class="protobuf.Vec3fProtos$Vec3f" code="sfvec3f"/>
<def class="protobuf.MFVec3fProtos$MFVec3f" code="mfvec3f"/>
<def class="protobuf.RotationProtos$Rotation" code="sfrotation"/>
<def class="protobuf.MFRotationProtos$MFRotation" code="mfrotation"/>
<def class="protobuf.MFStringProtos$MFString" code="mfstring"/>
</codes>
```

Mind the different handling of multifields in InstantIO and RSB. InstantIO uses Java arrays of the corresponding singlefield, while RSB uses dedicated multifield versions. So for InstantIO the multifields are named `[L{singlefield name};`, putting a square bracket and an L in front of the singlefield and appending a semicolon.

8 Slot file

A slot file (also called "sensor file") is a XML document with definitions to select specific sensor fields (slots) from the source. The option to name this file is `--slotfile`, or short `-s`:

```
--slotfile sensor.xml
```

The use of a slotfile tells VeniceHub to switch from dynamic slot creation to predefined slot creation.
Example for a slotfile:

```
<Venice>
    <simulator>
        <status type="sfstring"/>
        <car>
          <position type="sfvec3f"/>
          <velocity type="sfvec3f"/>
        </car>
    </simulator>
    <participant>
        <gaze type="sfvec3f"/>
        <expression type="mffloat"/>
    </participant>
</Venice>
```

In the above example VeniceHub will create the slots

```
simulator/status (sfstring)
simulator/car/position (sfvec3f)
simulator/car/velocity (sfvec3f)
participant/gaze (sfvec3f)
participant/expression (mffloat)
```

The first element (in the example named `Venice`, but can have any name) is the unique root element of the XML document, so it has to contain all other elements, according to the definition of a well-formed XML document. This root element contains now all elements for namespaces and slots. If an element has a `type` attribute, it will be parsed as a sensor slot of that type. If an element do not contain the `type` attribute, it will be parsed as a part of the namespace. The name of the element is used as the namespace and the slotlabel respectively.
Optionally it is possible to use the `name` attribute; in that case VeniceHub will use the value of the `name` attribute for namespaces and slotnames.
The next example defines the same slots as the first example, but uses a more verbose syntax (as in older versions of VeniceHub and VeniceIPC):

```
<Sources>
    <Sensor name="simulator">
        <Slot name="status" type="sfstring"/>
        <Namespace name="car">
            <Slot name="position" type="sfvec3f"/>
            <Slot name="velocity" type="sfvec3f"/>
        </Namespace>
    </Sensor>
    <Sensor name="participant">
        <Slot name="gaze" type="sfvec3f"/>
        <Slot name="expression" type="mffloat"/>
    </Sensor>
</Sources>
```

9 Format for VP strings

In VP mode VeniceHub receives via TCP socket strings that have to be in a format that is defined by a XML file (`--vpfile {filename}`). This XML file uses the same conventions as the XML file for predefining slots.
The fields have to be separated by a comma and a space. The first value has to be the checksum (Adler32). The string has to end with a newline character.
Multifields: A multifield has to start with an integer, which indicates the number of fields belonging to the multifield. The subfields in a multifield have also to be separated with a comma and a space.
Example: A MFVec2f with 3 fields has to be formated like:

```
3, 0.11, 0.12, 0.21, 0.22, 0.31, 0.32
```

It is possible to have arrays with zero fields.
Selfexplaning Example:

```
<Venice>
  <experiment>
    <data>
      <frameNumber type = "SFInt32"/>
      <headConfidence type = "SFFloat"/>
      <position type = "SFVec3f"/>
      <rotation type = "SFRotation"/>
      <title type = "SFString"/>
      <switchA type = "SFBool"/>
      <objectnames type = "MFString"/>
    </data>
  </experiment>
</Venice>
```

A received string can look like this:

```
    1868306011, 12, 0.1, 102.0, 100.3, 144.04, 0.1, 0.2, 0.3, 0.4, Test, true,
         2, Box, Sphere\n
```

Remember that the first value is the ckecksum.

It is allowed to use namespaces. Just concatenate the fields, as if there where no namespaces. Only the order is important.
Example:

```
<Venice>
  <sensor1>
    <position type = "SFVec3f"/>
    <frame type = "sfint32"/>
  </sensor1>
  <sensor2>
    <shape type = "sfstring"/>
  </sensor2>
</Venice>
```

And a resulting string:

```
942212455, 0.1, 0.2, 0.3, 45, Circle
```

The fields of the two namespaces `sensor1` and `sensor2` are just concatenated.

10 Protobuf Classes

This section describes the use of protobuf classes with RSB. First the path to the folder with the protobuf classes is needed. The relevant option is
`--protobuf {folderpath}`.
Then VeniceHub needs to know what XIO code belongs refers to what protobuf class. This is done with the option
`--xiocodes {filename}`
See section 7 how to create such a file.
This is sufficient for the tasks of:

- reading from disk and sending to RSB
- receiving from RSB and writing to disk

But if a conversion from InstantIO to RSB or vice versa is needed, then a third step is needed: A XML file that defines what InstantIO type maps to what RSB protobuf type. This file is given with the option `--classMatcher {filename}`. This takes a bit more work, see section 11 for more information.

10.1 How to create a custom RSB protobuf class file

See the proto files that are included. They are located in the protos folder of venice.lib. They should have the following structure:

```
package protobuf;
option java_package = "protobuf";
option java_outer_classname = "XXXProtos";
message XXX {
  // your values
}
```

Where XXX is the name of your class. The name of the file has to be XXX.proto.

Example for single field boolean:

```
package protobuf;
option java_package = "protobuf";
option java_outer_classname = "BoolProtos";
message Bool {
  optional bool value = 1;
}
```

Note: The assigned value is not an actual value, but an index (starting with 1). See google protocol buffers documentation for more information.

Example for multifield float:

```
package protobuf;
option java_package = "protobuf";
option java_outer_classname = "MFFloatProtos";
message MFFloat {
  repeated float value = 1 [packed=true];
}
```

Create a folder named `protobuf` and copy your proto files one level above.

Example:

```
create folder /myApplication/defs/protobuf
copy your proto files to /myApplication/defs/
```

Compile each proto file with protoc (get it from google and copy it in the folder where the proto files are located). The protobuf jar (e.g. protobuf-java-2.4.1.jar) has to be in that folder, too.

```
> protoc --java_out=. XXX.proto
```

This creates a XXXProtos.java in the protobuf folder. This source file can already be used in your source code. But for the use with venice.lib, further compiling is necessary.

Change directory to the protobuf folder, which should now contain the above mentioned java files created by protoc. Type the following command:

```
> javac -cp .;..\protobuf-java-2.4.1.jar XXXProtos.java
```

This creates all class files. venice.lib should recognize them now.

11 Class matching between RSB and InstantIO

What is matching good for? An protobuf class object, received by RSB, needs to be converted into an IIO class object. For example, if you have created a class called `protobuf.Vec3fProtos.Vec3f` as an counterpart to the `Vec3f` of InstantIO, it is necessary to tell venice.lib, how it can transfer the values.

This is done by XML file. See match.xml in the venice.lib main directory as an example. The structure is the following:

```
<matches>
  <match from="..." to="...">
    ...
  </match>
</matches>
```

The "from" attribute and the "to" attribute in the "match" element are defining what source class is matched to what target class. This has to be done for each of the two directions. In case of matching RSB and IIO, the relation between the two cases is NOT symmetric.

11.1 Example Float

You have created an protobuf class named MyFloat, which contains a single float value. You want to match it to the java native Float (java.lang.Float), because Floats can be send over InstantIO out-slots.

```
<match from="protobuf.MyFloatProtos.MyFloat" to="java.lang.Float">
  <constructor parameter="float"/>
  <getter name="getValue"/>
</match>
```

The constuctor element tells venice.lib, that the target class can be contructed with a public constructor. venice.lib assumes, that the constructor has the same name as the target class (according to java rules). It also tells, that this constructor takes one parameter of the primitive type float.

The getter element tells venice.lib, that the float value of the protobuf class can be retrieved with the method getValue().

The other direction don't work this way, because there are no public constructors for protobuf classes. Instead there are builders. venice.lib will automatically use builders, if the target class is a protobuf class. In this case a <methodpair> element is needed:

```
<match from="java.lang.Float" to="protobuf.MyFloatProtos.MyFloat">
  <methodpair getter="floatValue" setter="setValue" type="float"/>
</match>
```

The getter attribute tells venice.lib how to get the value from the Float. The setter attribute tells venice.lib how to set the value for your target class. venice.lib will then instanciate the builder class for this protobuf class, call the setter method and then commands the builder to build the instance.

11.2 Example for multifield classes

Let's say, your protobuf class from the last example is now a multifield, so it can store multiple float values. It is not possible to just take the singlefield class and make an array out of it, like you would do with java.lang types. For protobuf you have to create a complete new class. Let's call it protobuf.MyMultiFloatProtos.MyMultiFloat.

To match this to a Float array:

```
<match from="protobuf.MyMultiFloatProtos.MyMultiFloat" to="[Ljava.lang.Float
    ;" repeated="true">
  <constructor parameter="float"/>
  <getter name="getValue"/>
</match>
```

The most important difference is the attribute repeated="true". The name of the Float array follows java naming conventions (wrapping into "[L" and ";"). The rest is the same.

The other direction:

```
<match from="[Ljava.lang.Float;" to="protobuf.MyMultiFloatProtos.MyMultiFloat
    " repeated="true">
  <methodpair getter="floatValue" setter="addValue" type="float"/>
</match>
```

The setter method is here "addValue" - this is an important difference between singlefield and multifield builder classes. Singlefield builder classes are using the "set" notation, while multifield builder classes are using "add".

11.3 Example singlefield protobuf class with more than one value

Let's say you have created a protobuf version of the Vec3f class of InstantIO. It takes three float values. They are named X, Y and Z. So a match entry would look like this:

```
<match from="protobuf.MyVec3fProtos.MyVec3f" to="org.instantreality.InstantIO
    .Vec3f">
  <constructor>
    <parameter type="float" index="0"/>
    <parameter type="float" index="1"/>
    <parameter type="float" index="2"/>
  </constructor>
  <getter name="getX" index="0"/>
  <getter name="getY" index="1"/>
  <getter name="getZ" index="2"/>
</match>
```

For the constructor element we need now more information, so there are the parameter elements. Each parameter element tells venice.lib of what primitive type the constructor parameter is, and the order by index (beginning with 0).

The getter elements now need the additional attribut "index", so venice.lib can tell, to what constructor parameter the value goes.

The other direction is easy:

```
<match from="org.instantreality.InstantIO.Vec3f" to="protobuf.MyVec3fProtos.
    MyVec3f">
  <methodpair getter="getX" setter="setX" type="float"/>
  <methodpair getter="getY" setter="setY" type="float"/>
  <methodpair getter="getZ" setter="setZ" type="float"/>
</match>
```

11.4 Example for a multifield protobuf class with more than one value per field

Let's say you have created a multifield version of the above example:

```
<match from="protobuf.MyMultiVec3fProtos.MyMultiVec3f" to="[Lorg.
    instantreality.InstantIO.Vec3f;" repeated="true">
  <constructor>
    <parameter type="float" index="0"/>
    <parameter type="float" index="1"/>
    <parameter type="float" index="2"/>
  </constructor>
  <getter name="getX" index="0"/>
  <getter name="getY" index="1"/>
  <getter name="getZ" index="2"/>
</match>
```

And the other direction:

```
<match from="[Lorg.instantreality.InstantIO.Vec3f;" to="protobuf.MyMultiVec3
    fProtos.MyMultiVec3f" repeated="true">
  <methodpair getter="getX" setter="addX" type="float"/>
  <methodpair getter="getY" setter="addY" type="float"/>
  <methodpair getter="getZ" setter="addZ" type="float"/>
</match>
```

12 Runtime Commands

While VeniceHub is running it is possible to enter commands via keyboard:

12.1 config

Shows all parameters and options of the actual configuration.

12.2 h

Shows a short list of important commands.

12.3 msg

Turns on or off messages VeniceHub is showing on the console.

12.4 offset [{x}]

Shows or sets the offset for replay. All timestamps will be modified by this offset. This is used mainly to synchronize VeniceHub replay with an RPC linked application (e.g. ELAN). `x` has to be in milliseconds. Default value is 0.

12.5 p

Switches between pause and play (only if input source is Disk).

12.6 reset

Resets the replay while replaying from Disk. Will start from beginning of the file again.

12.7 savelag

If lag history is activated, this command will cause VeniceHub to save the collected lag data to file.

12.8 seek {timestamp}

Seeks for a timestamp (in milliseconds) while replaying from Disk.

12.9 skip {time} [ms | s | m | h]

Skips the given time while replaying from Disk. If the unit parameter is omitted, time will be interpreted as milliseconds. Use negative values to skip backwards.

12.10 time

Shows the timestamp of the last replayed data object. Will be modified by offset, if an offset was given.

12.11 xio

Shows the XIO codes VeniceHub is using. XIO codes are the tag names of XIO lines representing the data type. For example `sfstring` represents by default a singlefield string object. See section 7 for more information.

13 Troubleshooting

13.1 Multicast

On some operation systems (e.g. Linux) there can be problems with the multicast connection. An error message like the following will be thrown:

```
java.net.SocketException: bad argument for IP_MULTICAST_IF: address not bound to
    any interface
```

In this case, try the following command for starting VeniceHub:

```
java -Djava.net.preferIPv4Stack=true -jar VeniceHub.jar [{config file}]
```

13.2 Quit

If there is a fatal error that freezes VeniceHub and it doesn't react to `q`, or it freezes while trying to end all threads, try pressing `ctrl` and `c`.

14 How to setup some Venice scenarios

In this section the use of VeniceHub together with other Software is shown by some examples.

14.1 VeniceHub and Kinect

Scenario: In an experiment a Kinect Sensor is used to get body data. The data needs to be logged (so that it can be replayed for an annotion task).

Solution: The KinectServer `BodyBasics-D2D.exe` of the VeniceKinectServer repository will receive Kinect data and send it to VeniceHub. A definition of the structure of the data is necessary, lets assume it is called body.xml.

First start VeniceHub with

```
> java -jar VeniceHub.jar -i VP --vpport {port} --vpfile body.xml -o Disk -f
    bodydata.xio.gz
```

VeniceHub will wait until the KinectServer establishes a connection. For that start

```
> BodyBasics-D2D {port}
```

VeniceHub should show the message, that a connection is established. The data will now be logged to the file bodydata.xio.gz.

14.2 VeniceHub and ELANMod

Scenario: In an experiment with Kinect Sensor, Camera and Microphone VeniceHub was used to log the data from the Kinect. The structure of the Kinect data is defined in a slotfile named bodydata.xml and a X3D file named body.x3d was created for InstantPlayer to visualize the Kinect data.

Now the Audio needs to be annotated by listening to the audio, seeing the Video and the visualized Kinect data. Audio and Video will be replayed by ELAN, but the Kinect data can not be visualized by ELAN. For the visualization of the Kinect data the InstantPlayer will be used and for the replay of the Kinect Data VeniceHub will be used. Now the only problem is to synchronize the replay of VeniceHub and ELAN, so that the annotating person can easily listen to the Audio, watch the Video and see the visualization of the Kinect Data, all synchronized.

As an additional difficulty, the Video starts 1500 ms later than the data in the Kinect log file.

Solution: Run VeniceHub with

```
> java -jar VeniceHub.jar -i Disk -f kinectlog.xio.gz -o IIO --offset -1500 -
    s bodydata.xml
```

assuming the file with the logged Kinect data is named kinectlog.xio.gz and is in the same directory as VeniceHub.jar. VeniceHub will begin to replay the data into the IIO network, where the InstantPlayer picks up the data and visualizes it.

Now the modified ELAN (ELANMod) is started and as soon as ELAN has started with its replay, it sends seek commands to VeniceHub, so the latter will synchronize its replay.

ELANMod and VeniceHub will both use the default values for the connection. If those needs to be changed, use for VeniceHub the options

```
--rpcServerAdress {adress}
--rpcServerPort {port}
```

To set those values to the configuration of ELANMod, edit the config.properties of ELANMod and change the entrys

```
browserIP={adress}
browserPort={port}
```

If the offset between Video and Kinect data was not set right, it is easily changed during runtime with the `offset {ms}` console command of VeniceHub.

14.3 Translating IIO ↔ RSB

Scenario 1: A data source provides data only via InstantIO (for example KinectServers), but there is a data sink that accepts only data via RSB (for example a Python program - so far there is no implementation of InstantIO for Python).

Solution: Use VeniceHub as a data filter. In the easiest situation, only data types are used that are also available as protobuf types in the default protobuf folder that is included in the release version of VeniceHub.

Then the command will be

```
 > java -jar VeniceHub.jar -i IIO -o RSB
```

VeniceHub will silently load the default protobuf types from `./protobuf` and the default class-matching-definitions-file `match.xml`.

If there are for some reasons no protobuf types available, it is possible to use a workaround:

```
 > java -jar VeniceHub.jar -i IIO -o RSB --rsbtoXIO
```

In this case, VeniceHub will only use the String data type of RSB (that does not need any protobuf classes), convert the InstantIO data into an XIO line and write this complete line as a String object to RSB. The data sink can than parse the XIO line and retrieve all information about original data type, value, timestamp, namespace and sensor label.

Scenario 2: A data source provides data only via RSB, but there is a data sink that accepts only data via InstantIO (for example InstantPlayer).

Solution: Use VeniceHub as a filter. Easiest way is

```
 > java -jar VeniceHub.jar -i RSB -o IIO [--mcttl {n}]
```

assuming that the default protobuf classes are sufficient to cover the InstantIO data. If not, the workaround to avoid protobuf classes is

```
 > java -jar VeniceHub.jar -i RSB -o IIO --rsbStringsAreXIO
```

With the `--rsbStringsAreXIO` option, VeniceHub will parse Strings received via RSB in the same way as lines in a log file.

Important is the `--mcttl {n}` option, if the data needs to be received on a different machine. The value {n} is the number of routers between VeniceHub and the receiving machine (the value can be greater than that number, but not less).

14.4 Multiple data sources, multiple data loggers

Scenario 1: In an experiment a lot of different data sources are used and all data needs to be logged on different machines. The used network protocoll is InstantIO and all sources and sinks are connected in a LAN with one router.

Solution: Be sure that all sources are using multicast and that multicast is enabled on each machine. Start on each logging machine

```
> java -jar VeniceHub -f {filename} [-s {sensorfile}]
```

The usual `-i` and `-o` options are here omitted, because in this case they are identical to the default values.

A sensor file with sensor data definitions in XML format can be given, if only a sub set of the sensor slots has to be logged.

If a source is also an instance of VeniceHub (for example translating some data from RSB to InstantIO), it is necessary to set there the option `--mcttl 1`, so that the data package does not get discarded by the router.

15 Tables and Figures

List of Tables

1 Modes of operation 4
2 Some options 6
3 Supported datatypes 8

List of Figures