UNIVERSITE D'ORLEANS
Faculté des Sciences
LIFO

Laboratoire d'Informatique Fondamentale d'Orléans
4, rue Léonard de Vinci, BP 6759
F-45067 Orléans Cedex 2
FRANCE

Rapport de Recherche

www : http://www.univ-orleans.fr/SCIENCES/LIFO/

Net Juggler Guide

Jérémie Allard, Loïck Lecointre,
Valérie Gouranton, Emmanuel Melin
and Bruno Raffin

Université d'Orléans, LIFO

Rapport N^o **2001-02**

Net Juggler Guide

Jérémie Allard, Loïck Lecointre
Valérie Gouranton, Emmanuel Melin and Bruno Raffin
Laboratoire d'Informatique Fondamentale d'Orléans (LIFO),
Université d'Orléans, Orléans, FRANCE

email: gouranton|melin|raffin@lifo.univ-orleans.fr

15th June 2001

Contents

Introduction

All the way through this book we assume the reader has some experience with VR Juggler. If not, refer to VR Juggler documentation (www.vrjuggler.org).

This book is also about clusters. We define a cluster as a set of computing nodes (or hosts) connected by a network, where each node supports a single system image, but the whole set of nodes does not. A set a linux PCs connected by a ethernet network is a cluster, but not a SGI Onyx.

Net Juggler is a software laying on top of VR Juggler that turns a cluster where each node supports VR Juggler into a single VR Juggler image machine. In other words, from the user's point of view it (almost) does not make any difference to run a VR Juggler application on a cluster, a single PC or a SGI Onyx (from the operational point of view and not from the performance point of view).

A very high-quality multi display projection or active stereo display requires the different video signals to be genlocked (video signal synchronization). Net Juggler does not include any support for genlock. If required, use appropriate hardware and/or software for genlocking the video signals.

Chapter 1 is about installing Net Juggler and running a first application. Chapter 2 goes more in the details about how to write configuration files for a cluster and how to make sure a VR Juggler application can run without troubles on a cluster. The chapters 4 and 3 are for readers interested in the design and the implementation of Net Juggler. The appendix A introduces the SoftGenLock library that implements a sofware genlock and enables quad buffer page flipped stereo without requiring any specific hardware.

Chapter 1

Getting Started Guide

1.1 Hardware Requirements

1.1.1 Graphics Cards

Net Juggler does not require any specific graphics card. In particular, because Net Juggler implements a software swaplock (swap buffer synchronization), graphics cards do not have to support swaplock. Most of todays common 3D accelerated graphics cards will ensure satisfactory results.

A very high-quality multi display projection or active stereo display requires the different video signals to be genlocked (video signal synchronization). Net Juggler does not include any support for genlock. If required, use appropriate hardware and/or software for genlocking the video signals. The SoftGenLock library presented in appendix A provides a sofware genlock and enables quad buffer page flipped stereo without any specific hardware.

1.1.2 Cluster Nodes

Net Juggler runs a copy of the VR Juggler application on each node of the cluster. Thus, if computation precision are not identical on each node, data may become incoherent. Using a cluster with identical nodes guarantees that this problem does not occur.

1.1.3 Network

Any kind of network can be used, provided that a communication API supported by Net Juggler is available (currently MPI) and that the performance is sufficient.

Net Juggler uses synchronization barriers and data communication instructions. Barriers are mainly used for the swaplock. Because only input events are sent over the network, bandwidth should not be a limiting factor.

Communication and synchronization time add extra latency in the main loop and thus can affect interactivity. We tested Net Juggler on a 4 node cluster with

- MPI over TCP/IP with an Ethernet network (10 Mbits/s),
- MPI over TCP/IP with a Fast Ethernet network (100 Mbits/s),
- MPI over GM with a Myrinet network (2 Gbits/s).

Performance was acceptable with the Ethernet network. With the faster networks extra time induced by communications and synchronizations was typically a few hundreds of microseconds. This is not significant compared to the tens of milliseconds required for a frame.

1.2 Software Requirements

1.2.1 Operating System

Net Juggler should support any operating system that VR Juggler supports. This include IRIX, Linux, Windows, Free BSD and Solaris.

At the moment we only tested Net Juggler with Linux and Windows. Please contact us if you successfully compile and run Net Juggler on an other OS.

1.2.2 VR Juggler

Net Juggler uses VR Juggler to run a copy of the application on each node. Thus, VR Juggler should be installed on each node.

Note that VR Juggler should be patched (patch included in the Net Juggler distribution) so that Net Juggler can be installed.

1.2.3 Graphics API

Net Juggler should support any graphics API that VR Juggler supports. This includes OpenGL and Performer.

The present version supports OpenGL and Performer, but swaplock support is not yet available for Performer.

1.2.4 Communication Library

Net Juggler is designed so that it can easily be ported on top of various communication libraries. Currently only MPI is supported. Thus, MPI should be installed on your cluster.

MPI is a widely available library ported to almost any kind of network. MPICH is the most common MPI implementation (www-unix.mcs.anl.gov/mpi/). In particular MPICH provides a MPI implementation over TCP/IP.

For specific networks higher performance MPI implementations may be available. Here is a non exhaustive list of high performance MPI implementations:

- MPI/Gamma supports various megabit and gigabit ethernet cards (www.disi.unige.it/project/gamma/).
- For Myrinet networks:
  - MPI/gm (www.myri.com) is the vendor provided implementation and is ported on various operating systems.
  - MPI/HPVM (www-csag.ucsd.edu/projects/hpvm.html) is a high performance implementation for Windows.
  - MPI/BIP (lhpca.univ-lyon1.fr) is a high performance implementation for Linux.

1.3 Installing VR Juggler

This section is a quick overview of the VR Juggler installation procedure.

We assume you use VR Juggler 1.0.

You need to download the VR Juggler source distribution `vrjuggler-1.0.0.tar.gz` from www.vrjuggler.org.

To install VR Juggler and test a sample application execute the following commands:

```
% tar -xvf <vrjuggler-1.0.0.src.tar.gz>
% ln -s vrjuggler-1.0.0.src vrjuggler
% cd vrjuggler
```

```
% patch -u -p 2 -i ../netjuggler-distrib/patch/vrjuggler-1.0.0.patch # you can
skip this to test VR Juggler, but it will be required to install Net Juggler.
% autoheader
% autoconf
% ./configure
% gmake
% export VJ_BASE_DIR=/???/vrjuggler/instlinks # also put this line on a config file
(like ~/.bashrc)
% cd samples/ogl/cubes
% gmake
% ./cubes $VJ_BASE_DIR/share/Data/configFiles/simstandalone.config
```

If an error occurs please refer to "VR Juggler Getting Started Guide".

Now you are ready to install Net Juggler.

1.4 Installing Net Juggler

For the moment Net Juggler is only distributed in a source form that you should have downloaded (`netjuggler-distrib.tar.gz`).

Until required modifications are made to VR Juggler distribution, Net Juggler can only be installed on a patched version of VR Juggler. The patch is distributed with Net Juggler. To apply this patch you need a VR Juggler source distribution.

1.4.1 Decompression

Once you have downloaded Net Juggler, unpack it in the directory you want to install it:

```
% tar -xzvf <netjuggler-distrib.tar.gz>
```

If your TAR version does not support unpacking gzipped tar files, execute instead:

```
% gunzip <netjuggler-distrib.tar.gz>
% tar -xvf <netjuggler-distrib.tar>
```

After the decompression a new directory named `netjuggler-distrib` should have been created. This directory is referred by <netjuggler_base>.

1.4.2 Patching VR Juggler

In the <netjuggler_base>/patch directory you should find files called `vrjuggler-distrib.patch`. Choose the patch corresponding to you VR Juggler distribution. If the corresponding file does not exist please update your Net Juggler distribution, or refer to chapter 4 for a description of the modifications that must be applied to VR Juggler.

Go to VR Juggler source directory and apply the patch:

```
% cd <vrjuggler source directory>
% patch -u -p 2 -i <netjuggler_base/patch/vrjuggler-distrib.patch>
```

Now you need to recompile VR Juggler:

```
% make
% make install
```

1.4.3 Compiling Net Juggler

Net Juggler uses a simple compilation process that does not yet support customization.

To compile Net Juggler all you need to do is to invoke `make` in the Net Juggler directory:

```
% cd <netjuggler_base>
% make
```

This should create the following files:

- `libNetJuggler.a`: main Net Juggler library
- `libNetJuggler_MPI.a`: MPI binding for Net Juggler

Few test programs are also compiled:

- `netapi_mpi_tests` : Test MPI performances. This is a standard MPI program
- `net_kernel_tests` : Simple Net Juggler test. See below for executing Net Juggler applications
- `net_input_tests` : Similar to net_kernel_tests, but using a keyboard input called "NetKeyboard" (declared in cluster.netkey.config).

1.4.4 Setting Environment Variables

Net Juggler requires the `NJ_BASE_DIR` environment variable to be instantiated to Net Juggler base directory <netjuggler_base>.

You need to add the following line in your `~/.bashrc` file for a personal installation, or in a global configuration file like `/etc/profile` if you have root access and want Net Juggler to be available to all users:

```
% NJ_BASE_DIR=<netjuggler_base> ; export NJ_BASE_DIR
```

Net Juggler includes a little utility called `juggler-config` used to easily compile VR Juggler applications. To ease access to this command add a symbolic link to `juggler-config` from a directory that is in your `PATH` :

```
% ln -s <netjuggler_base>/juggler-config <bin_directory>
```

If you want to activate Net Juggler by default, you can define `USE_NETJUGGLER` environment variable to `yes` near the line defining `NJ_BASE_DIR`:

```
% USE_NETJUGGLER=yes ; export USE_NETJUGGLER
```

1.4.5 Configuration Files

VR Juggler uses configuration files to control input and output devices like trackers, displays.... This is the same on a Net Juggler cluster, but the files should include extra informations like the name of the node (or host) the tracker is connected to.

Updating configuration files for your cluster is detailed in chapter 2. For running your first VR Juggler application on a cluster, the distribution includes two sets of configuration files (almost) ready to use (see `netjuggler_base/Data/config`). These files are written for a cluster with 4 nodes named `pc1`, `pc2`, `pc3` and `pc4`:

- `cluster.*.config`:
  - `cluster.base.config`: Basic configuration. Must be included. Set the time device on `pc1`.
  - `cluster.netconnect.config`: Configure `pc1` to open the TCP port 4451 for the dynamic reconfiguration of the cluster with VjControl.

  - `cluster.displays.config`: Set the views displayed on each node of the cluster:
    * `pc1`: front display
    * `pc2`: right display,
    * `pc3`: left display,
    * `pc4`: floor display.
  - `cluster.wand.mixin.config`: Set a simulated wand using `pc1`'s keyboard.

- `sim.*.config`: Correspond to the VR Juggler files for running an application in simulator mode. All inputs are on the first node of the cluster and all nodes display the same view.

To have a fully functional configuration, you must at least have a keyboard and a display for `pc1` You can of course customize the configuration files for an other cluster configuration (see chapter 2).

We provide a small utility program so you do not have to go through all configuration files to update host names according to your cluster configuration. Just fill out the `hosts.txt` file in `netjuggler_base` directory:

```
@pc1@=host1
@pc2@=host2
@pc3@=host3
@pc4@=host4
```

where host# is the host name of a node in the cluster, and run the command (in the Net Juggler directory):

```
% make config
```

For nodes `pc3` and `pc4`, you can put unused host names if you do not have the corresponding nodes on your cluster.

If you want to test Net Juggler with only one machine, give the first host the machine's name and the second host the same name but with `:2` appended:

```
@pc1@=host1
@pc2@=host1:2
@pc3@=toto
@pc4@=toto
```

1.5 Compiling an Application

The following section deals with compiling a VR Juggler application by hand or with a makefile.

Ideally any VR Juggler application should compile and run on a Net Juggler cluster without any modification. In fact reality is a little bit different and a few things need to be done. Refer to chapter 2 for more details.

In this section we detail how to compile the "cubes" sample application of VR Juggler that we ported as an example (`<netjuggler_base>/samples/vrjuggler/cubes`).

1.5.1 Compilation by Hand

If your application only contains few source files you can build it by directly invoking the compiler. All you need to do is to use the `juggler-config` tool that returns the compilation arguments needed for Net Juggler. Possible options are:

```
Usage: juggler-config [OPTIONS] [LIBRARIES]
Options:
  [--version]
```

```
    [--libs]
    [--cflags]
  Libraries:
    vrjuggler
    netjuggler
    mpi
```

To compile your application, use the following command:

```
% gcc -o <app_exe> <source_files> ‘juggler-config --libs \
--cflags<juggler_options>‘ <app_options>
```

where:

<app_exe> is the executable file name

<source_files> are the application source files

<juggler_options> are the Net/VR Juggler options

<app_options> are the application specific options and libraries

If the environment variable USE_NETJUGGLER is set to yes, juggler-config default options are netjuggler mpi, vrjuggler otherwise.

For example, if you want to compile the "cubes" sample application using Net Juggler with MPI use:

```
% gcc -o cubes cubes.cpp cubesApp.cpp ‘juggler-config --libs \
--cflags netjuggler mpi‘
```

Note that juggler-config works exactly like gtk-config from GTK+.

1.5.2 Using a Makefile

Have a look to <netjuggler_base>/samples/vrjuggler/cubes for a sample Makefile for Net Juggler.

All you need to do is to use juggler-config to define compiler flags and linker options. This can be done by adding the following lines to your Makefile:

```
CFLAGS= $(CFLAGS) ‘juggler-config --cflags netjuggler‘
LIBS=$(LIBS) ‘juggler-config --libs netjuggler‘
```

Note that to easily switch between compiling for VR Juggler or for Net Juggler, you have two possibilities:

- Use juggler-config with the vrjuggler or netjuggler option.
- Set the variable environment USE_NETJUGGLER to yes or no and do not specify any netjuggler or vrjuggler option to juggler-config.

1.6 Running an Application

This section is about launching a VR Juggler application on your cluster. It depends on the communication library used by Net Juggler. Currently Net Juggler only supports MPI (see chapter 2 for more details).

1.6.1 MPI Binding

A Net Juggler application is launched with the standard MPI command. Generally MPI implementations include a `mpirun` script. The arguments of the `mpirun` script must include the VR Juggler application you want to execute, how many processes you want to execute, generally one per node, and the VR Juggler configuration files updated for your cluster.

For example, `mpirun` is used to launch the "cubes" application on 4 nodes using the `cluster` configuration files (see section 1.4.5) as follow:

```
cd  <netjuggler_base>samples/vrjuggler/cubes
mpirun -np 4 cubes \
    $NJ_BASE_DIR/Data/config/cluter.base.config \
    $NJ_BASE_DIR/Data/config/cluster.netconnect.config \
    $NJ_BASE_DIR/Data/config/cluter.displays.config \
    $NJ_BASE_DIR/Data/config/cluter.wand.mixin.config
```

To run the application in simulator mode, change the configuration files:

```
cd <netjuggler_base>samples/vrjuggler/cubes \
mpirun -np 4 cubes \
       $NJ\_BASE\_DIR/Data/config/simstandalone.config
```

Here are some essential `mpirun` arguments :

`-np <num>`: Number of process to launch.

`-machinefile <file>`: Configuration file containing the cluster node list.

`-nolocal` : Do not launch a process on the local host.

Chapter 2

Programmer's Guide

2.1 Introduction

This chapter describes how to modify a VR Juggler application so that it can run on a cluster with Net Juggler, and how to update the configuration files.

Net Juggler is a software harness for running VR Juggler applications on a cluster. Ideally, a VR Juggler application should run without any modification on a Net Juggler cluster. For the moment, Net Juggler is not yet fully integrated into VR Juggler and minor modifications are required for a VR Juggler application to be runable.

Net Juggler cluster configuration is achieved by using the same configuration files than VR Juggler. These files are enriched with extra information needed by Net Juggler (e.g. the name of the node the tracker is connected to).

2.2 Checklist

This part is brief checklist of the modifications required to run a VR Juggler application on a Net Juggler cluster. More detailed informations are provided in the next sections.

To run a VR Juggler application on a Net Juggler cluster it is necessary to:

1. Call `vjKernel::parseArg` at the beginning of the main procedure

2. Modify the Makefile to use `juggler-config` (see chapter 1).

3. Check that the application retrieves all input data through VR Juggler input devices. If not, modify the application.

4. Modify the configuration files to include host informations.

2.3 Argument Parsing

In the main procedure, a call to `vjKernel::parseArg` must be inserted just after the kernel is created:

```
vjKernel* kernel=vjKernel::instance(); // this line should already
                                       // be present in main
kernel->parseArg(&argc,&argv); // THIS LINE MUST BE ADDED
```

Note that `argc` and `argv` must not be used before `parseArg` call.

2.4 Makefile

The Makefile has to be modified so that the Net Juggler libraries be linked to the application at compile time (see chapter 1).

2.5 Cluster Configuration Chunks

The VR Juggler configuration system is based on a set of files containing "chunks". These chunks describe the configuration of each system component. To run a VR Juggler application on a Net Juggler cluster, the configuration files should be modified (directly editing the files or using VjControl) to include cluster related extra informations.

2.5.1 The Host Parameter

Each configuration chunk must include a `Host` parameter. The `Host` specifies the cluster node the chunk is applied to.

A `Host` parameter can take one of the following values:

- `""` or `"All"` : chunk applied to each host of the cluster
- `"hostname"` : chunk applied only to the host specified

For example a `User` chunk that concerns each host and a `FrontDisplay` chunk that concerns only the host `grappe7` should be defined as:

```
JugglerUser
  Name "User"
  Host { "All" }
  ...
  end
DisplaySurface
  Name "FrontDisplay"
  Host { "grappe7" }
  ...
  end
End
```

The following general rules can be observed to define the `Host` parameters:

- A display surface applies to one host only.
- An input device applies to one host only.
- An input proxy is a special case treated in the next section.
- All other chunks apply to all hosts.

2.5.2 Input Proxies

A VR Juggler application never directly accesses an input device but uses an intermediate proxy device.

Net Juggler extends this approach to define a new class of input proxies, the "shared" proxies. On a cluster, an input device, a wand for example, is connected to one node, but the data must be broadcasted to all other nodes. When a "shared" proxy is encountered, Net Juggler knows that the data retrieved from that proxy must be broadcasted to each node of the cluster. This solution is elegant as it requires no modification of the application code.

A shared proxy chunk is similar to a standard proxy chunk except that the chunk name is prefixed by `Shared`.

The `Host` parameter of a shared proxy chunk is interpreted as the source of the shared data. Thus, the `Host` parameter must be the same as the `Host` parameter of the associated input device.

For example, the following chunks define a shared proxy for the `TimeSystem` input device running on `pc1` (see section 2.7.2 for more details about `TimeSystem`).

```
vjincludedescfile
  Name "timesystem.desc"
  end
TimeSystem
  Name "TimeDevice"
  Host { "pc1" }
  end
SharedAnaProxy
  Name "Time"
  Host { "pc1" }
  device { "TimeDevice" }
  unit { "0" }
  end
End
```

Note that standard input proxies can be useful on a cluster. For example, a keyboard can be associated to a node only to change the viewport of the display associated with that node. In that case, the keyboard proxy should not be shared.

2.6 Use VR Juggler Input Devices

Net Juggler runs a copy of the VR Juggler application on each node of the cluster. To keep data coherency between all application instances, Net Juggler intercepts data inputs and broadcast them to each node. Net Juggler intercepts only the data the application retrieves through VR Juggler input device. A VR juggler application could directly get a random number or a time bypassing VR Juggler. In this case, Net Juggler is not able to guarantee data coherency. So, you should check that your VR Juggler application retrieves all its input data from VR Juggler input devices. If not, you must modify the application to use VR Juggler input devices. You can implement your own VR Juggler input devices or use the ones provided with the VR Juggler or Net Juggler distributions.

The Net Juggler distribution includes a time input device described in the following section. It can be used has a pattern to develop other input devices, a random generator input device for example. Please refer to VR Juggler documentation for more informations about input devices.

2.7 The Time Input Device

The time input device included in the Net Juggler distribution is an analog input that returns the amount of time taken by the last frame. It can be used to retrieve a time data for physical simulations or other computations that modify application states based on a time delay.

2.7.1 `TimeSystem`

The `TimeSystem` input is implemented as a VR Juggler analog input device. Use a `vjAnalogInterface` to access it:

```
vjAnalogInterface mTime; // put this in your application class
```

In the `vjApp::init()`, `mTime` must be initialized and named:

```
mTime.init("Time"); // put this in your application init() method
```

When you need to obtain the time taken by the last frame, usually in the `preFrame` method, you have to use:

```
float dtime=mTime->getData(); // dtime contains the last frame duration
                              // in seconds (clamped to 1)
```

Then use `dtime` in your code to update the application states.

2.7.2 Configuration Chunks for `TimeSystem`

`TimeSystem` is a standard VR Juggler analog input device that requires configuration chunks that must be included in a configuration file.

A chunk defines the `TimeSystem` device and an other chunk the associated proxy. For a VR Juggler system, the chunks are:

```
vjincludedescfile
  Name "timesystem.desc"
  end
TimeSystem
  Name "TimeDevice"
  end
AnaProxy
  Name "Time"
  device { "TimeDevice" }
  unit { "0" }
  end
End
```

Once modified for a Net Juggler system, we obtain the chunks presented in section 2.5.2.

Chapter 3

Design Guide

Net Juggler was developed with the following goals:

- No modification should be required to run a VR juggler application on a cluster.
- Launching the application should not require the user to access each node.
- Configuring the cluster should not require the user to access each node.
- All VR Juggler features, like run-time reconfiguration or performance data collection, should be available on a cluster.
- Net Juggler should be as transparent as possible such that any new feature that could be added in future VR Juggler releases, should be also available on a cluster, ideally with no porting effort.
- The required modifications to VR Juggler code should be minimal.
- Net Juggler organization should respect the VR Juggler organization (micro-kernel architecture).
- Net Juggler should ensure high performance executions. In particular, communication and synchronization costs should be minimized.
- Net Juggler should include a software swaplock support for the clusters that do not have hardware swaplock support.
- No cluster node should have a master position for better scalability.

We present in this chapter how Net Juggler was designed to meet these goals.

3.1 General Design Choices

3.1.1 Parallelization Paradigm

To run a VR Juggler application on a cluster we adopted a simple parallelization paradigm: each node of the cluster runs its own copy of the application with its own local parameters, like the viewport for instance. Obviously, input devices are not duplicated. Thus to ensure data consistency across the different copies, input events are broadcasted to each node. This parallelization can easily be hidden fom the user, it is scalable and ensures that the amount of data to communicate is small. The main drawback is that it can lead to redundant computations. Future works will address this problem.

3.1.2 Sharing Inputs

The user of a VR application needs different input devices to interact in real-time with the application, like gloves, keyboards, trackers... VR Juggler collects these inputs and forward them to the application. The approach is the same with Net Juggler except that a given input device is connected to one given node only. Consequently, Net Juggler must get the inputs from the device, and broadcast the collected data to each node of the cluster.

Proxies and Inputs

Let us explain more specifically how Net Juggler gets the input data and how it broadcasts them.

VR Juggler manages each input through a driver (`vjInput` class). This driver is connected to a proxy (`vjProxy` class) that forwards the data to the application.

We could use specific drivers to transmit data. We would associate a server input driver to the node the device is connected to, and a client input driver for the other nodes. The main advantage is that it is very easy to add new drivers in VR Juggler. We just need to instantiate a client class and a server class for each kind of input driver. The drawback is that every single device driver would require a client and a server input driver. This may be pretty laborious.

We did not adopt this solution, but we translated it at the proxy level. Instead of having client and server input drivers, we have client and server proxies. Proxies provide an abstraction of input drivers and thus their number is limited and should not increase significantly in the future. This approach only requires to modify the `vjProxy` class in VR Juggler so that we can derive it. Also note that a VR Juggler proxy stores a pointer to its input driver. It is used to detect if the driver is connected or not. For a server proxy this is the same. For a client proxy the pointer is set to null.

3.1.3 Configuration Management

System Configuration

The system configuration is very important in VR Juggler. It can be controlled by files given when starting the program, or by requests sent during the execution from VjControl. The system configuration is seen like a list of chunks, each chunk having some informations about a part of the system (display, input,...).

One goal of Net Juggler is to use only one global configuration for the whole cluster, allowing at the same time to have nodes with different configurations (different viewports for example). We add a "Host" parameter to a configuration chunk that can be equal to "All" or to a node name. It points out that the considered chunk applies to all nodes of the cluster or only to the specified node.

The chunk associated to each couple of a client/server proxy is renamed by taking the regular VR Juggler proxy name prefixed with `"Shared"`. The parameter `"Host"` has then a different semantics: it points out the node that runs the server proxy, all the other nodes having a client proxy.

Processing Configuration Chunks

Configuration chunks are stored in a data base on each node before being transmitted to VR Juggler. We want each node to know the whole cluster configuration to avoid to centralize configuration informations on one specific node or to have to handle scattered chunks when the user asks for the configuration.

Each node has a configuration filter to select the chunks that must be applied locally.

Dynamic Configuration

To dynamically configure VR Juggler, VjControl connects to VR Juggler through a TCP connection and sends configuration requests to `vjConfigManager`.

We extend this concept to Net Juggler. VjControl can connect to any node of the cluster running a configuration server. Configuration requests are intercepted and broadcasted to all nodes before being stored in each local data base and forwarded to the configuration filter.

Note that we keep two open port per node. The "old" VR Juggler port opened by the environment manager and the Net Juggler port. The global cluster configuration can be obtained and modified by connecting VjControl to the Net Juggler port. Through the VR Juggler port only local node informations can be retrieved. It is convenient for debugging purpose or to retrieve performance data. However this connection should not be used to modify the node configuration.

3.1.4 Communications

Communications must take place to broadcast configuration requests and input data. For performance purpose these data transfers must be carefully managed.

Streams

We use and extend the classical stream paradigm to represent data communication between nodes. There is one stream by server proxy and by configuration server. A stream is associated to a specific node source and can have several destination nodes. Each stream is identified by a unique id number and can be created, deleted or modified at run-time. The abstraction level provided by the streams hides the actual data movements that take place at a lower level.

Messages

Data communications take place only once per frame. When a node writes into a stream, it builds a message containing the data and appends it to the buffer of pending messages. When the communication actually takes place each node broadcasts its buffer to each other node. This collective communication operation is usually called an allgather.

Configuration events can take place at any time and cause buffers to have an unpredictable size. The adopted semantics for the allgather requires all nodes to know the size of the messages they will receive. When the allgather is executed, it sends input data and a special message indicating the size of the reconfiguration data. If this size is different from 0 a second communication step is triggered to send the list of the reconfiguration messages.

Network API

To ease portage to different communication libraries, Net Juggler has a communication interface hiding the library used.

3.1.5 Starting the Application

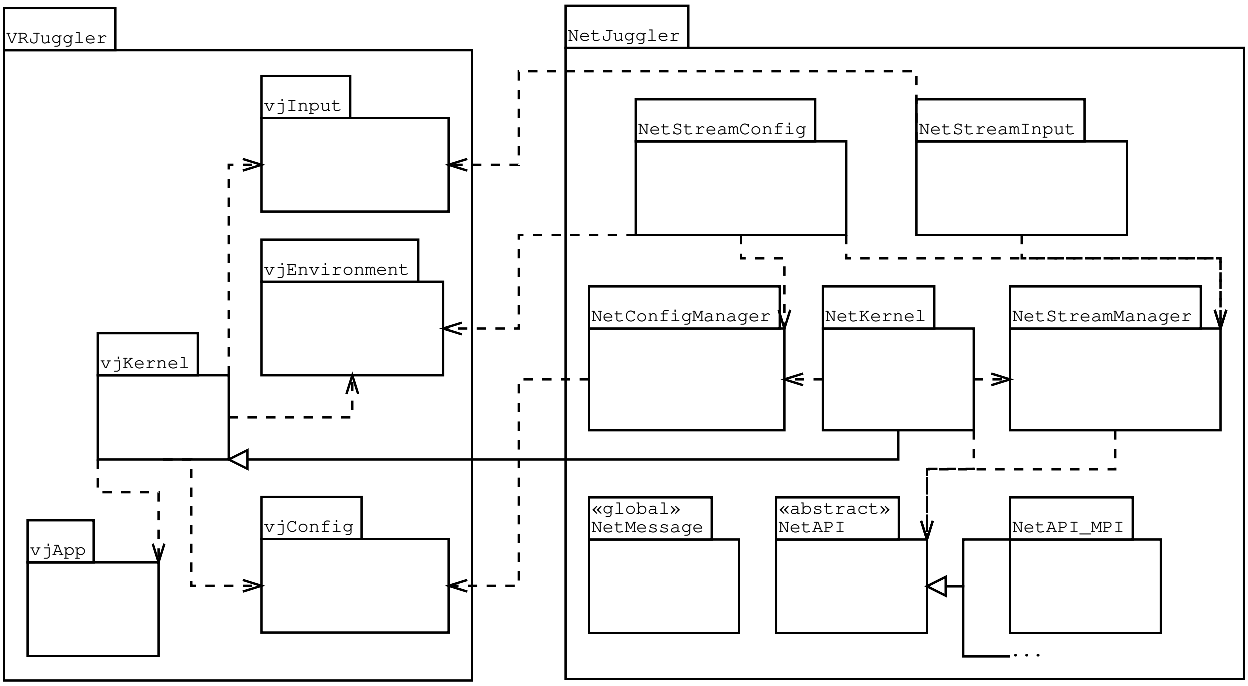
VR Juggler triggers the following sequence of actions when launched: the config files are loaded, next the kernel starts and only after the application is associated to the kernel. Though not really used, it should also be possible to change the application at run-time.

Net Juggler reuses the same sequence of actions. To ensure that the application is started on each node with the same context (same configuration and same input data), a synchronization barrier is required.

3.2 Net Juggler Architecture

Net Juggler architecture is organized as follow[1]:

[1]UML diagrams done with dia `www.lysator.liu.se/~alla/dia/`

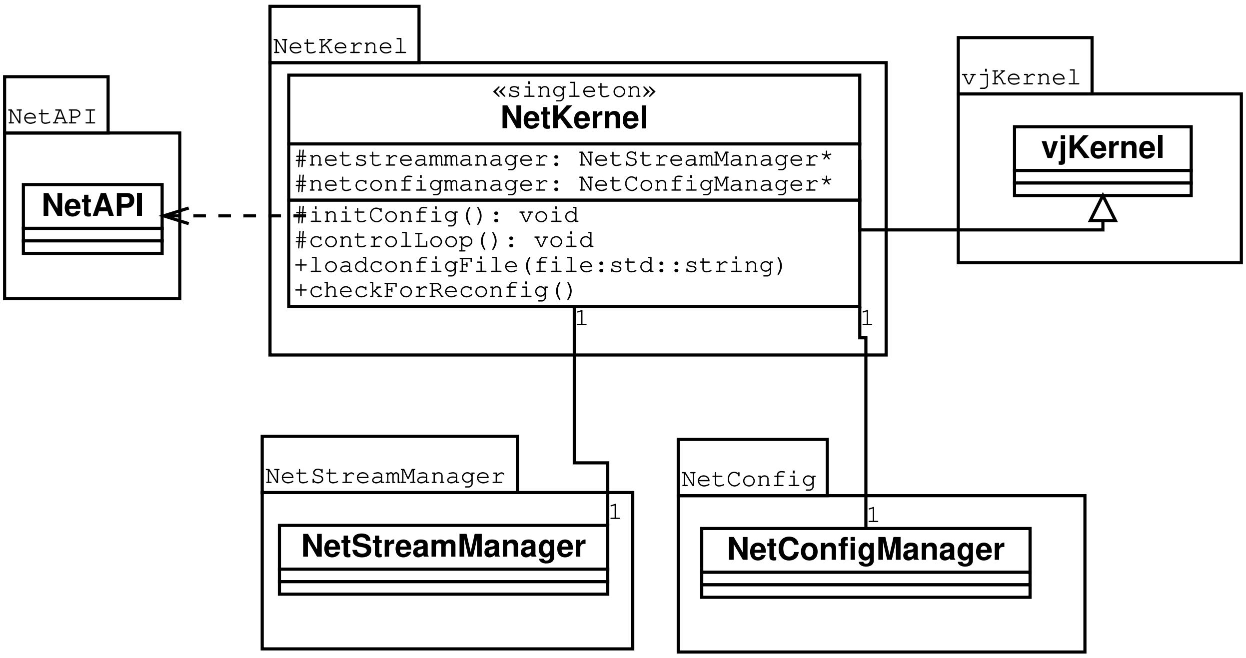


The role of the different modules is:

- `NetKernel` is Net Juggler kernel. It derives from VR Juggler kernel (`vjKernel`).
- `NetConfigManager` stores the cluster configuration and the pending chunks.
- `NetStreamManager` is responsible for stream management.
- `NetInputStream` gathers the classes related to shared inputs, i.e. the client and server versions of VR Juggler proxies.
- `NetConfigStream` is in charge of:
  - the connection to VjControl ;
  - the stream of configuration requests.
- `NetMessage` contains the classes that define the message object.
- `NetAPI` is an abstract interface defining the communication primitives.

3.2.1 NetKernel

UML Specification



Description

- `NetKernel` : Modify the VR Juggler kernel.

  - `netstreammanager`: Pointer to the `NetStreamManager` initialized in the `initConfig` method.
  - `netconfigmanager`: Pointer to the `NetConfigManager` initialized in the `initConfig` method.
  - `initConfig()`: Overload the `vjKernel` function. Call `vjKernel::initConfig()` and initializes the `NetAPI`, the `NetStreamManager` and the `NetConfigManager`.
  - `controlLoop()`: Main kernel loop. Similar to `vjKernel::controlLoop()`, but also activate the `NetStreamManager` and the `NetConfigManager`.
  - `loadConfigFile`(): Overload the `vjKernel` method. Send pending configuration chunks to the `NetConfigManager`.
  - `checkForReconfig()`: Overload the `vjKernel` method. Filter the pending reconfiguration chunks provided by `NetConfigManager` and then call the VR Juggler `checkForReconfig()` method.

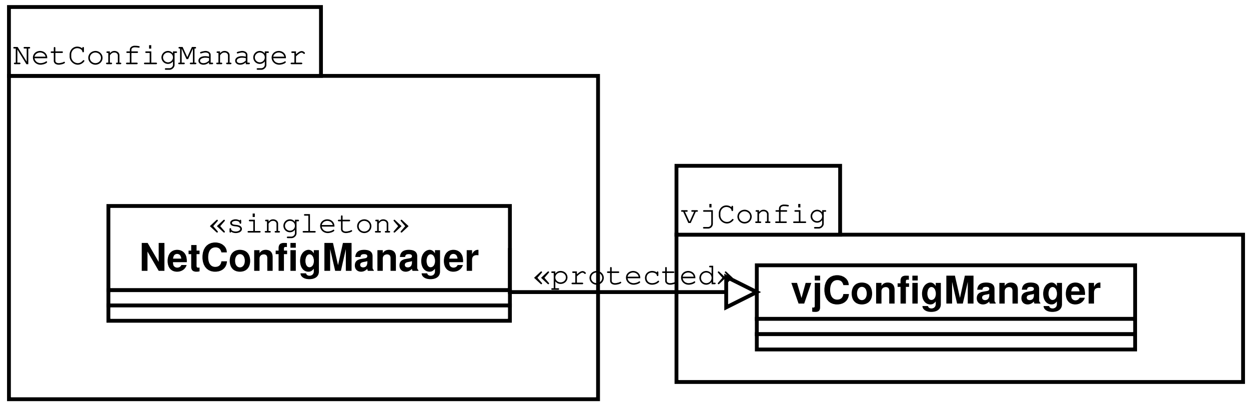
Remarks

The derivation of the `vjKernel` class allows to add the functionalities required by Net Juggler. This approach enforces modularity but requires the modification of the `vjKernel` and the singleton system (see section 4).

The `NetAPI` can be seen as a manager. It is initialized and controlled by the `NetKernel`. It does not interact directly with other managers to respect the micro-kernel organization.

3.2.2 NetConfigManager

UML Specification



Description

- `NetConfigManager` : Stores the current cluster configuration and the pending configuration requests.

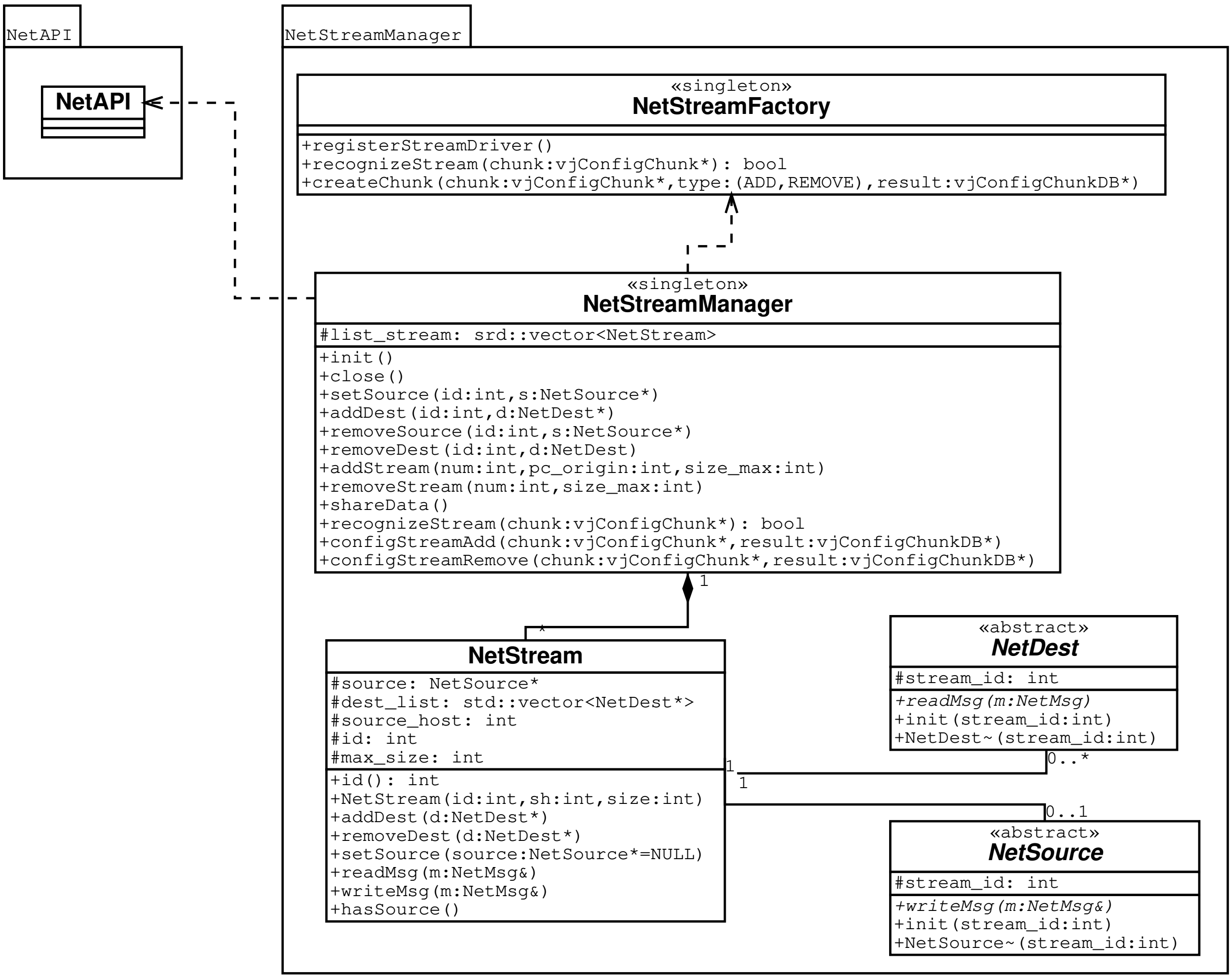
Remarks

Each node must store the current cluster configuration to answer VjControl requests.

`NetConfigManager` has the same methods than `vjConfigManager` but the former holds the cluster configuration and the latter the local node configuration. `NetConfigManager` does not filter the pending chunks not to create a dependence with the `NetStreamManager`, which would be in opposition with the micro-kernel architecture. Filtering takes place in the `NetKernel:checkForReconfig()` methods.

3.2.3 NetStreamManager

UML Specification



Description

- `NetStreamFactory`: Create Streams.
  - `registerStream()`: Record a new stream.
  - `recognizeStream()`: Check if a chunk corresponds to a stream.
  - `createChunk()`: Create the needed chunks for adding or removing a stream.
- `NetStreamManager`: Manage the streams.
  - `list_stream`: List of the streams used.
  - `init()`: Initialization.
  - `close()`: Close the class.
  - `setSource()`: Set a stream source node (overwrite the previous source if already set).
  - `addDest()`: Add a destination node to a stream.
  - `removeSource()`: Remove the stream source node.

  - `removeDest()`: Remove a destination node to a stream.
  - `addStream()`: Add a new stream.
  - `removeStream()`: Remove a stream.
  - `shareData()`: For each stream, the data written in the stream by the source node are sent to the destination nodes.
  - `recognizeStream()`: Check if a stream is already recorded.
  - `configStreamAdd()`: Add a stream with `addStream()` and create the associated chunks with `createChunk()`.
  - `configStreamRemove()`: Remove a stream with `removeStream()` and create the associated chunks with `createChunk()`.

- `NetStream`: Define the stream object.
  - `source`: Source object (server) of the stream.
  - `id`: Stream id.
  - `max_size`: Stream max size.
  - `source_host`: Rank of the source node.
  - `dest_list`: List of destination objects (clients) of the stream.
  - `id()`: Return the stream id.
  - `sourceHost()`: Return the rank of the source node.
  - `addDest()`: Add a destination node to `list_dest`.
  - `removeDest()`: Remove a destination from `list_dest`.
  - `hasSource()`: test if the stream source is set.
  - `setSource()`: Set the source node of the stream.
  - `writeMsg()`: Write a message into the stream.
  - `readMsg()`: Read a message from the stream.

- `NetDest`: Stream destination.
  - `stream_id` : Stream id.
  - `readMsg()`: Read a message from the stream.
  - `init()`: Initialization.

- `NetSource`: Stream source.
  - `stream_id`: Stream id.
  - `writeMsg()`: Write a message into the stream.
  - `init()`: Initialization.

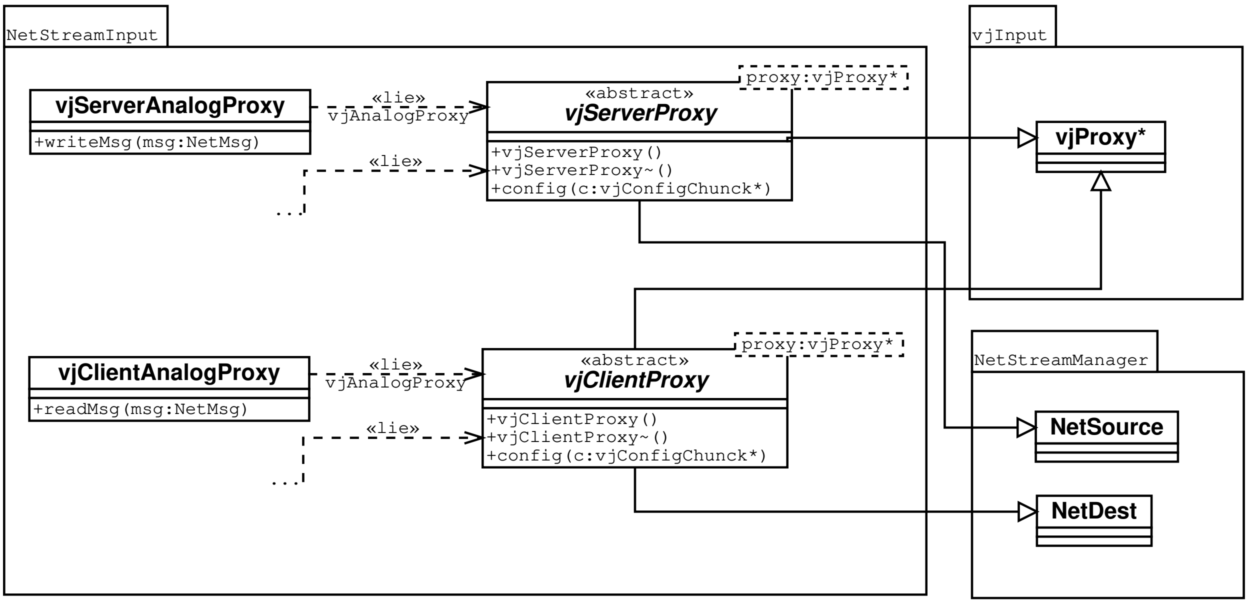
Remarks

The class `NetStreamFactory` is similar to a VR Juggler factory.

The `NetStreamManager` manages streams and is also responsible for filtering stream configuration chunks.

3.2.4 NetInputSream

UML Specification



Description

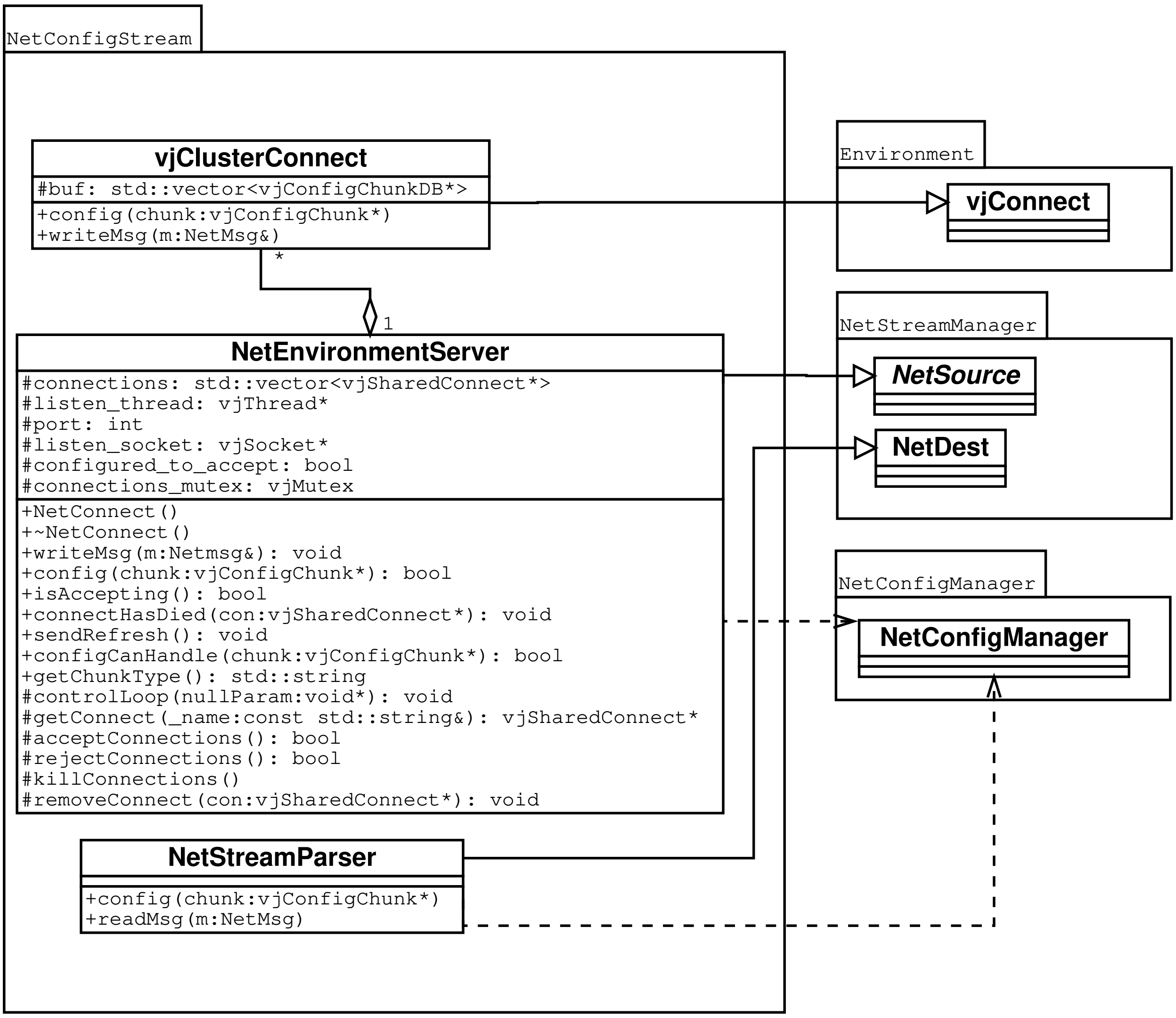
- `vjServerProxy`: Connected to an input device driver like a `vjProxy`, but also responsible for writing the retrieved data in an associated stream.
  - `config()`: Set the proxy.
- `vjClientProxy`: Retrieve input device data from a stream. The associated input device driver runs on a distant node. Its data are intercepted and written in the stream by the `vjServerProxy` running on the distant node.
  - `config()`: Set the proxy.
- `vjServerAnalogProxy`: Example of a `vjServerProxy` instantiation.
  - `writeMsg()`: Write the data received from the input driver into the associated stream.
- `vjClientAnalogProxy()` : Example of a `vjClientProxy` instantiation.
  - `readMsg()`: Read the data from the stream.

Remarks

The `vjServerProxy` and `vjClientProxy` classes are templates that can be used for any type of proxy (`vjAnalogProxy` in this example).

3.2.5 NetConfigStream

UML Specification



Description

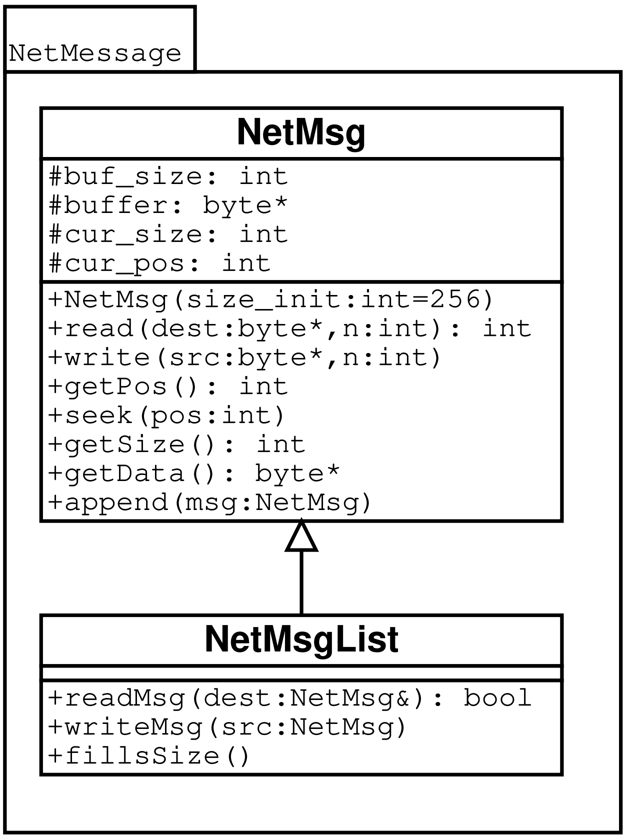
- `vjClusterConnect`: Manage the connection to VjControl for the cluster configuration.
  - `buf`: Buffer containing the received pending requests.
  - `config()`: Initialize the connection.
  - `writeMsg()`: Write the pending requests in the configuration stream.
- `NetConfigStreamParser`: Receive the pending configuration requests and transmits them to `NetConfigManager`.
  - `config()`: Initialization.
  - `readMsg()`: Read the received the pending configuration requests from the configuration stream and forward them to `NetConfigManager`.

- NetEnvironmentServer: Manage the list of connections and the associated sockets.

  - connections: List of connections.
  - listen_thread: Thread listening on connection port.
  - port: Port number.
  - listen_socket: Passive socket listening on port.
  - configured_to_accept: Boolean set to true if NetConnect can accept connections.
  - connections_mutex: Used to control concurrent accesses.
  - writeMsg(): Call the method writeMsg() of each vjClusterConnect.
  - config(): Configuration.
  - is_Accepting(): Test if a connection is possible.
  - connectHasDied(): Test if the connection is still active.
  - sendRefresh(): Tell VjControl it should refresh its image of the cluster configuration.
  - configCanHandle(): Test if the chunk can be added to the list of chumks to be processed.
  - getChunkType(): Return chunk type.
  - controlLoop(): Control the thread main loop.
  - getConnect(): Return a connection.
  - acceptConnections(): Test if connections can be accepted.
  - rejectConnections(): Deny connections.
  - killConnections(): Kill all connections.
  - removeConnect(): Remove a connection.
  -

3.2.6 NetMessage

UML Specification



Description

- `NetMsg`: Message handling.
  - `buf_size`: Size of the buffer containing the message.
  - `buffer`: Pointer to the buffer.
  - `cur_size`: Current message size.
  - `cur_pos`: Current position in the buffer.
  - `read()`: Read `n` bytes in the buffer from `cur_pos`.
  - `write()`: Write `n` bytes in the buffer from `cur_pos` and update `cur_size`.
  - `getPos()`: Return `cur_pos`.
  - `seek()`: Set `cur_pos` to a given position.
  - `getSize()`: Return `cur_size`.
  - `getData()`: Return the starting address of the message stored in the buffer (usually the starting address of the buffer).
  - `append()`: Append a given message at the end of the message already stored in the buffer. Update `cur_size` and `cur_pos` accordingly.
- `NetMsgList`: Higher level message handling methods.

- `readMsg()`: Read the next message stored in the buffer.
- `writeMsg()`: Add a message in the buffer.
- `fillSize()`: Add a ghost message in the buffer. This method is used for message padding.

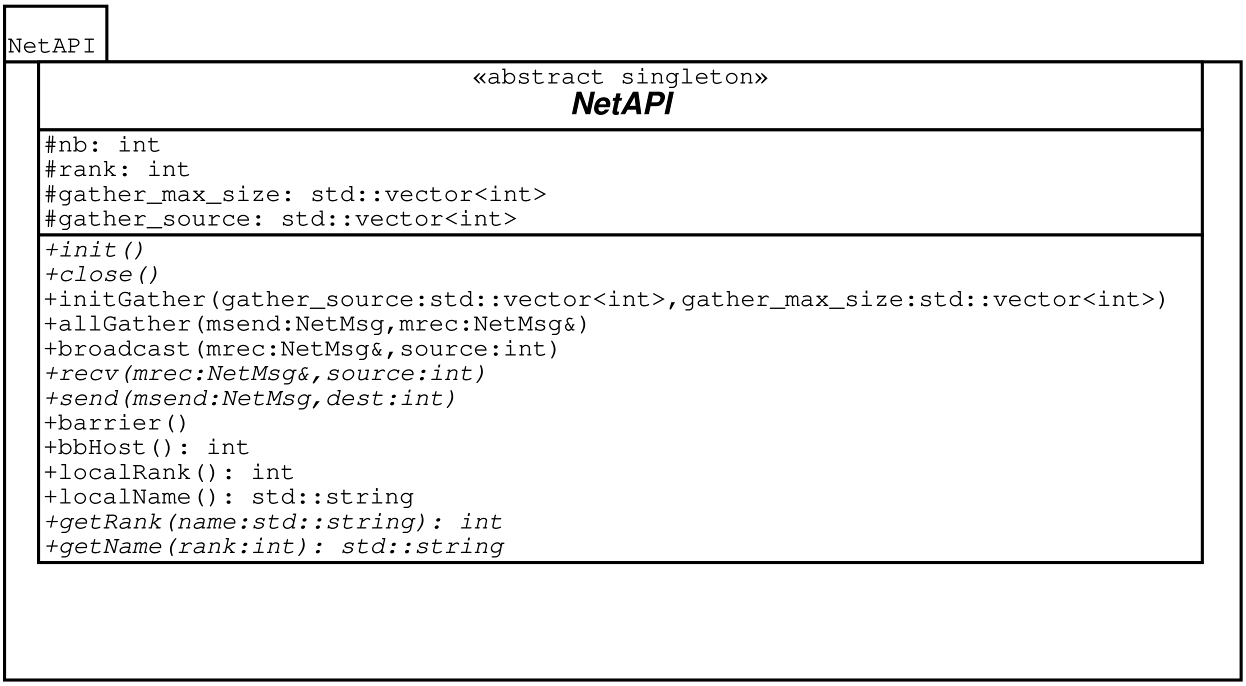
Remarks

The buffer is the space reserved to store a message. A message can be a concatenation of smaller messages. The methods of `NetMsgList` hides the details of reading and writing a message from a list (or concatenation) of messages.

Message copies can significantly affect communication performance, in particular for large messages (what is considered large depends on the network). Specific protocols are developed to avoid messages recopies. Not to limit the benefits of such protocols, Net Juggler should also avoid message copies, even if message size is typically small (a few hundreds of Kbytes).

3.2.7 NetAPI

UML Specification



Description

- `NetAPI`: Define the network interface used by `NetJuggler`
  - `nb`: Number of nodes.
  - `rank`: Node rank.
  - `gather_max_size`: Maximum size of the messages sent by the sources for the allgather.
  - `gather_source`: Source list for the allgather.
  - `init()`: Initialize the communication API.
  - `close()`: End.

- `initGather()`: Set the source nodes and the message maximum size parameters for the `AllGather()`.
- `allGather()`: Each node receives the message sent by each source node (equivalent to a gather followed by a broadcast).
- `broadcast()`: The source node sends a message to each other node.
- `recv()`: Wait until a message is received from the source node.
- `send()`: Send a message to the destination node.
- `barrier()`: Synchronization barrier between all nodes.
- `nbHost()`: Return the number of nodes.
- `localRank()`: Return the local node rank.
- `localName()`: Return the local node name.
- `getRank()`: Return the rank from a given node name.
- `getName()`: Return the node name from a given node rank.

Remarks

Splitting the allgather in an initialization function `InitGather` and a communication function `AllGather` allows to avoid repeating unnecessary initializations.

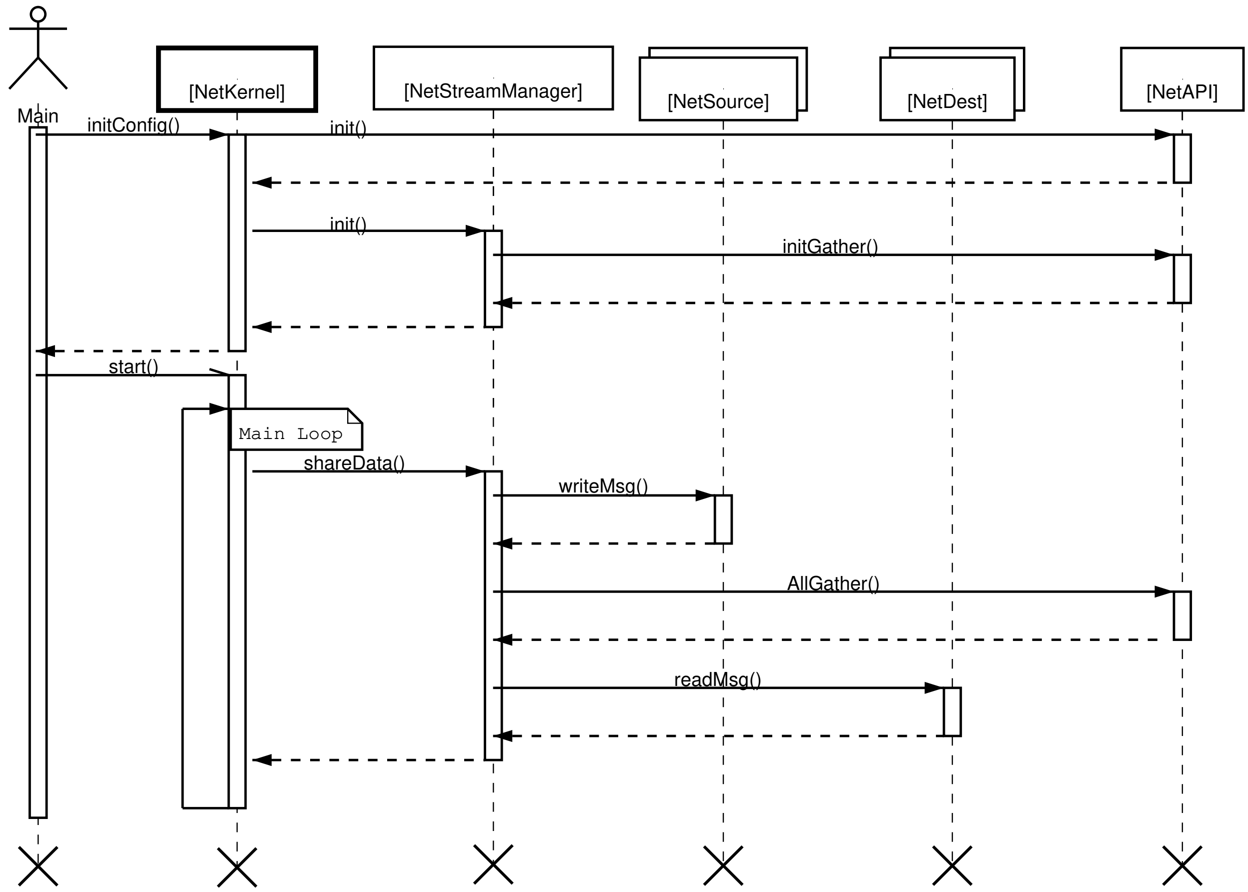
The `Init` function may contain the code necessary to build a data base storing the correspondence between node ranks and node names. This data base is then accessed using the `getRank` and `getName` methods.

3.3 Sequence Diagrams

This section shows the calling order of the main Net Juggler methods.

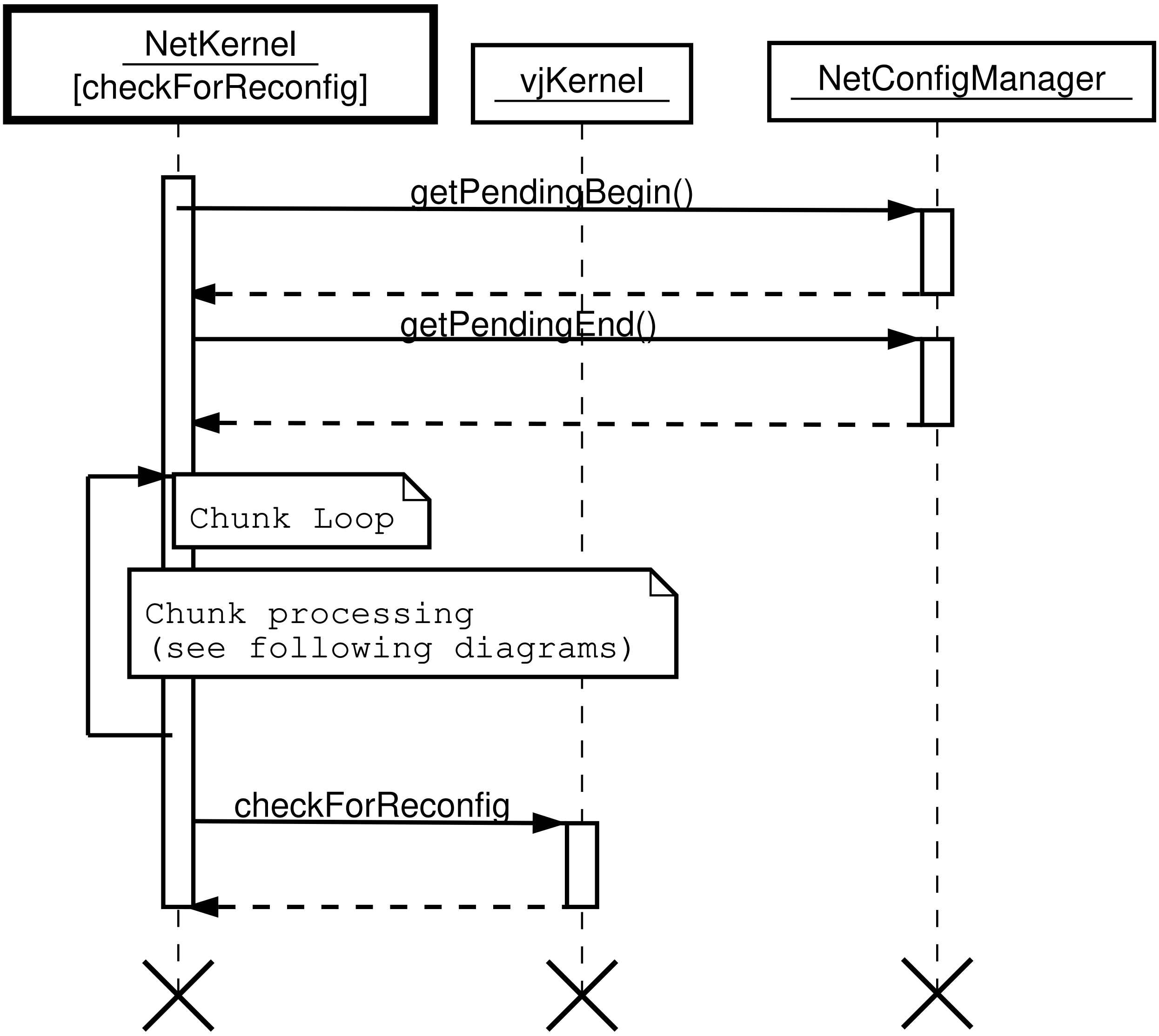
3.3.1 Data Exchange

The `main` function of the application launches `InitConfig()` that sets the `NetAPI` and the `NetStreamManager`. The `NetStreamManager` initializes the `AllGather()` parameters. At each iteration of the main loop, the kernel calls the `shareData()` function that is divided in 3 steps. Each source node stores in a buffer the concatenation of the messages to send. The allgather communication takes place. Each destination node reads the received data.



3.3.2 Configuration Chunk Handling

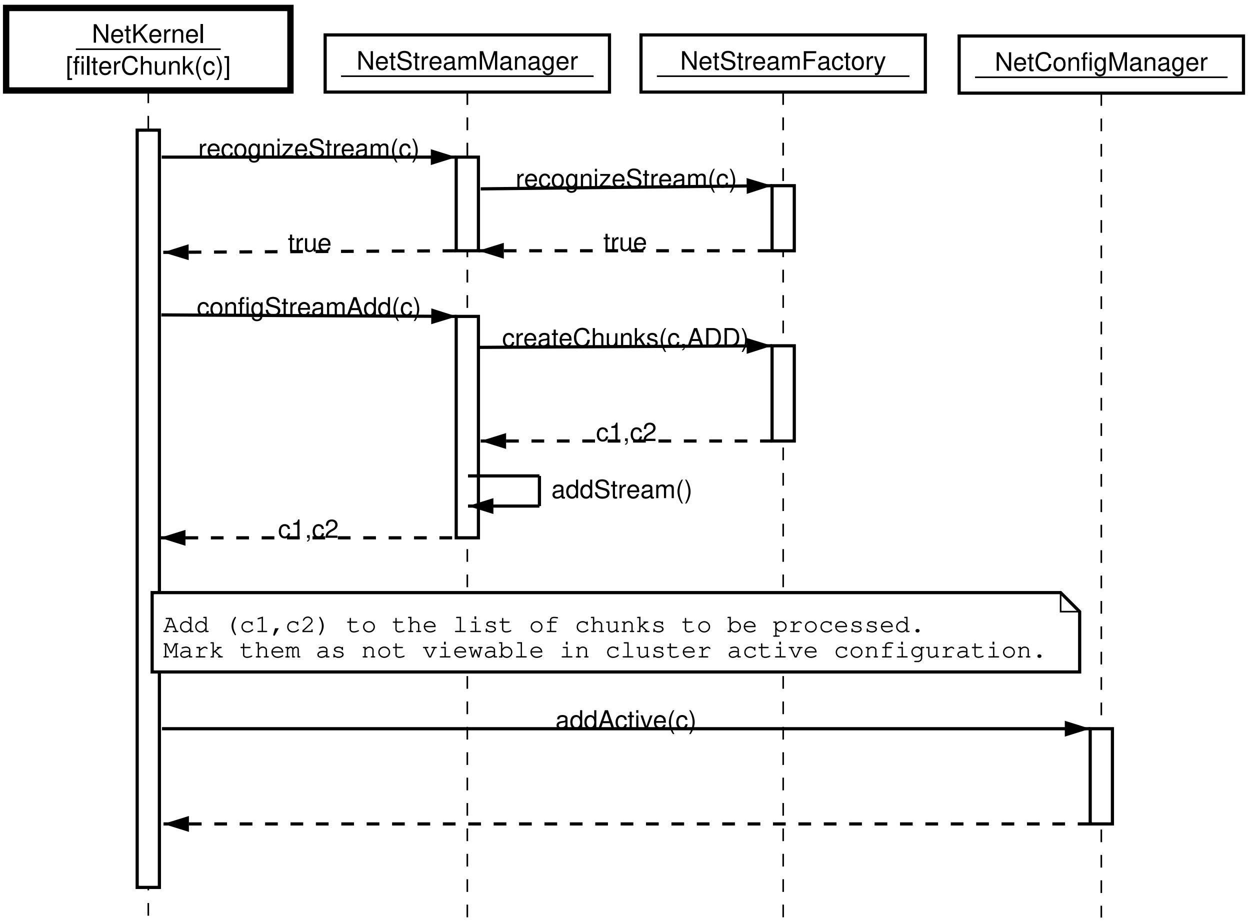
Before configuration chunk are passed to VR Juggler, they pass through a configuration filter implemented in `NetKernel::checkForReconfig`. The filter detects stream chunks and filters out non local chunks depending on the host parameter. The following diagram shows the filter main loop:



For each chunk processed, 3 cases are possible. They are described in the following sections.

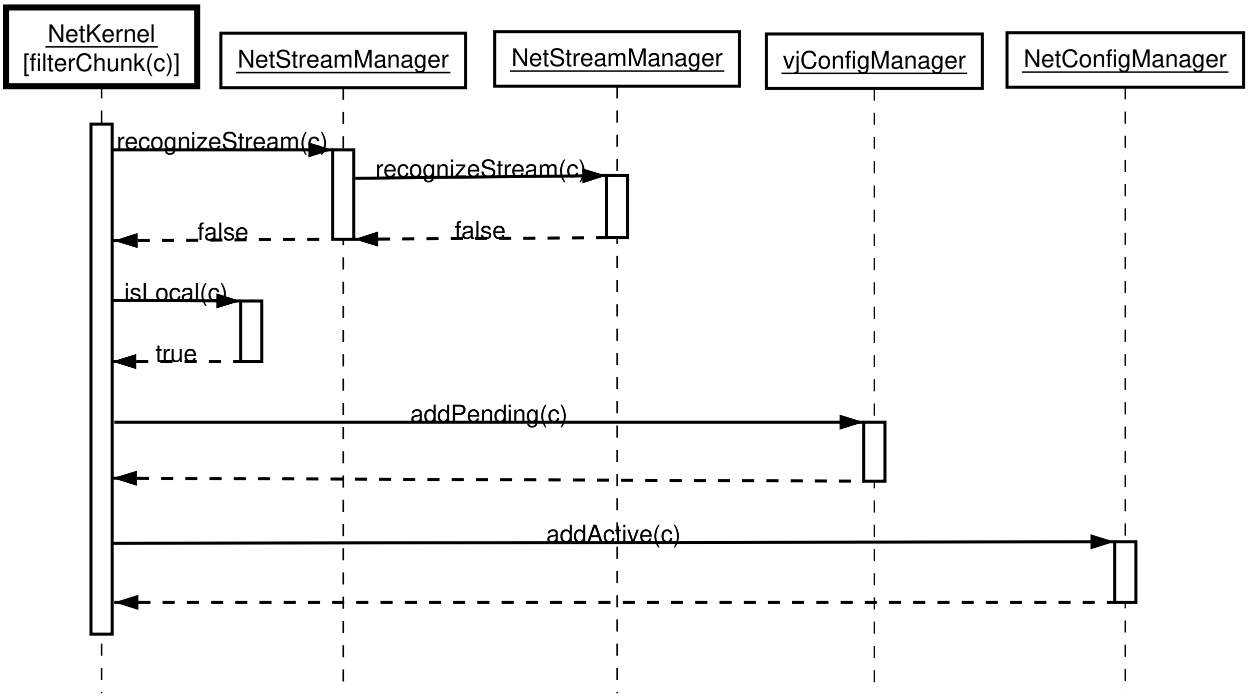
Stream Chunk Processing

Stream chunks are associated to shared objects, for example a shared proxy. The configuration filter must first recognize this kind of chunk. The chunk is then passed to `NetStreamManager` to create the stream and generate two chunks, one for the client and one for the server. Theses chunks are added to the list of chunks to be processed. They are not directly passed to VR Juggler as they may not be local. For example the server is only instanciated on one host. The initial stream chunk is next added to the current cluster configuration.



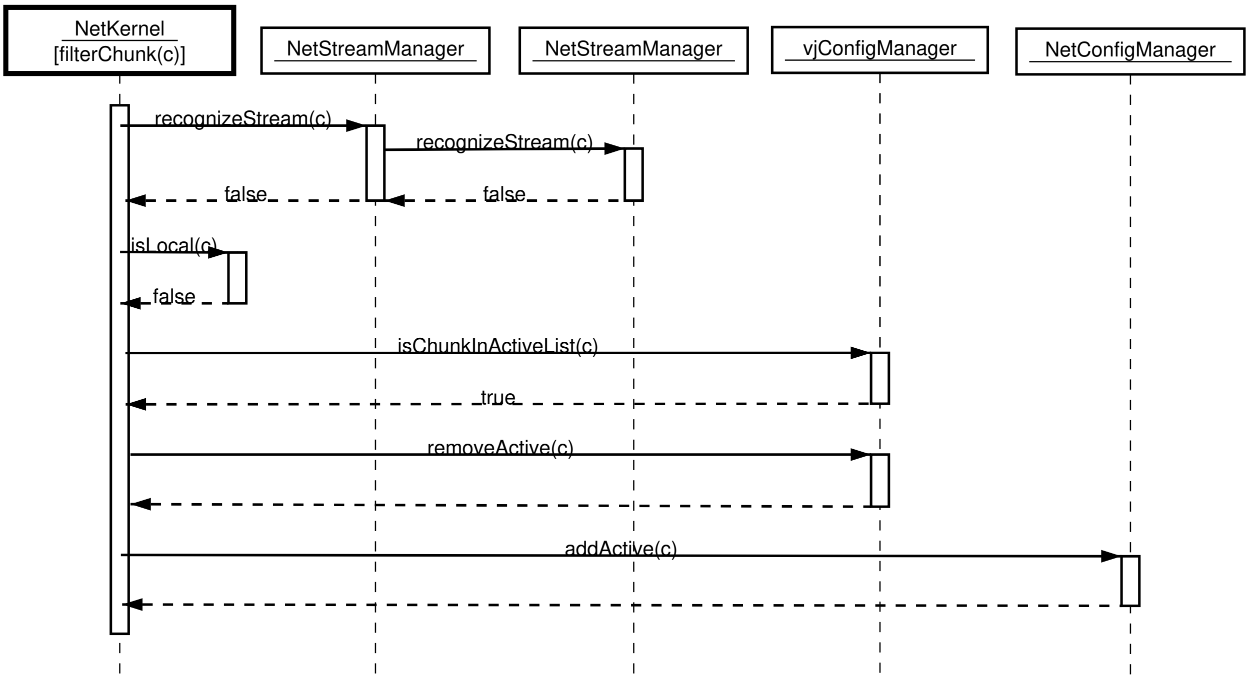
Local Chunk Processing

To detect a local chunk, the filter uses the `isLocal` method, passing as argument the host parameter of the chunk. If the host parameter corresponds to `"All"` or to the local host name, it is added to VR Juggler's `vjConfigManager` pending chunk list. It is also added to the current cluster configuration.



Non Local Chunk Processing

A non local chunks is a chunk that failed the preceding tests. If this chunk name also appears in the current local VR Juggler configuration, this means that it moved to a distant node. It must be removed from the the local VR Juggler configuration. It is next added to the current cluster configuration.



Chapter 4

Implementation Guide

4.1 VR Juggler modifications

The section details the modifications VR Juggler requires to support Net Juggler. A patch that should prevent you from doing it by hand is included in the Net Juggler distribution (see section 1). These modifications do not affect the VR Juggler overall architecture. In the future, they may be directly included in the main VR Juggler distribution.

4.1.1 Derived Classes

Net Juggler tries whenever possible to derive VR Juggler classes instead of modifying directly the VR Juggler code. This requires the modified methods to be declared `virtual`, which is not always the case (no one ever thought that `vjKernel::checkForReconfig` could be overloaded).

The affected classes are:

- `vjKernel`
- `vj*Proxy`
- `vjConnect`

4.1.2 Chunk Type Checking in the vj*Proxy Class

The `vj*Proxy:config` method checks Chunk types. Because Net Juggler defines two new proxy types (client proxy and server proxy), the test must be modified accordingly. For example the `vjAnalogProxy` test must be modified as:

```
vjASSERT(((std::string)chunk->getType()) == "AnaProxy"
      || ((std::string)chunk->getType()) == "AnaClientProxy"
      || ((std::string)chunk->getType()) == "AnaServerProxy");
```

4.1.3 VR Juggler 1.0 Related Issues

VR Juggler 1.0 implementation leads to compilation problems when proxies and input devices are registered externally to VR Juggler. These problems are related to the template classes `vjDeviceConstructor` and `vjProxyConstructor` constructors that are defined in the `.cpp` files and not in the `.h`.

To compile Net Juggler you need to move the corresponding code in the `.h` files (`Input/InputManager/vjProxyFactory.h` and `Input/InputManager/vjDeviceFactory.h`).

The `InputManager` produces an error when a proxy is added without an attached input device. Net Juggler requires such a possibility because client proxies are not attached to an input device. The methods `vjInputManager::add*Proxy` in the `Input/InputManager/InputManager.cpp` file must be modified to accept stupified proxies.

On Win32 systems, the `vjTimeStamp` class is not implemented, hence Net Juggler timer can not work. You must add an empty `diff` method in `vjTimeStampNone` (file `Performance/vjTimeStampNone.h` to be able to compile:

```
//:  returns 0.0
inline float diff (const vjTimeStampNone& t2) const {
return 0.0;
}
```

4.1.4 Calling a Derived Class Constructor

We describe the method we chose to call a derived class constructor without explicitly calling it to improve code modularity. For sake of clarity we concentrate on Net Juggler kernel creation, but this method also applies to other classes, for example the `NetAPI` and `NetAPI_MPI` classes.

The `vjKernel` class is called a "singleton" because only one instance of that class can be created. This is achieved by hiding the call to the constructor in a `intance` method. This method creates the kernel instance if it does not already exist, and returns the instance address.

Because a VR Juggler application needs a pointer to the kernel instance it has a pointer to the kernel initialized with the `instance` method:

```
vjKernel* kernel = vjKernel::instance();
```

On a Net Juggler cluster, an instance of the `NetKernel` is required instead. A solution would be to modify each application to call the `instance` method of the `NetKernel` class:

```
vjKernel* kernel = NetKernel::instance();
```

To avoid such a modification, we take advantage of the singleton system and modify its implementation (see `Utils/vjSingleton.h` for VR Juggler original singleton implementation and below page 38 for the modified version). The idea is the following: instead of calling the `vjKernel` constructor, the method `instance` uses a pointer `sInstanceConstructor` to an active constructor. This pointer points to `vjKernel`'s constructor if the `vjKernel` class is not derived, and to `NetKernel`'s constructor if `NetKernel` derives from `vjKernel`.

The `NetKernel` class is a "derived singleton". It has a specific `instance` method to set the `sInstanceConstructor` base pointer. Because the `sInstanceConstructor` pointer must be set before the instance of `NetKernel` is created, the `NetKernel` class has a static variable `isRegistered`. This variable initialization changes the constructor pointed by `sInstanceConstructor`.

Singletons are used for other classes, like `vjDeviceFactory`, so it is important that our implementation stay compatible with VR Juggler singleton system: if no derived class is provided the `sInstanceConstructor` should point to the base constructor. For that goal, `vjKernel` initializes the `static` variable `sInstanceConstructor` with `vjKernel`'s constructor.

We now have to make sure that `isRegistered` is initialized after `sInstanceConstructor` to set `sInstanceConstructor` to the expected constructor if a derived class is provided. We force a proper order by initializing `sInstanceConstructor` with a pointer assignment while `isRegistered` is initialized with a function call. Compilers first initialize simple variables, for example those without constructors or function calls, and then complex ones. All compilers we are working with respect this initialization order, but others may not. Please contact us if you encounter such a situation.

We also defined an "abstract singleton". An abstract singleton differs from a normal singleton because it can not be instantiated if no not-abstract derived class is defined (see page 38). The abstract singleton is required by `NetAPI`, the Net Juggler class defining the network interface.

```
#define vjSingletonHeader( TYPE )                    \
protected:                                           \
   typedef TYPE *vjSingletonPtr;                     \
   typedef TYPE vjSingletonBase;                     \
   typedef vjSingletonPtr vjSingletonConstructor();  \
   static vjSingletonConstructor *sInstanceConstructor; \
   static vjSingletonPtr constructor( void );        \
```

```
public:                                         \
   static TYPE* instance( void )

#define vjDerivedSingletonHeader( TYPE )        \
protected:                                      \
   static vjSingletonPtr constructor();         \
   static bool registerSingleton();             \
   static bool isRegistered;                    \
public:                                         \
   static TYPE* instance( void )

#define vjAbstractSingletonHeader( TYPE )       \
protected:                                      \
   typedef TYPE *vjSingletonPtr;                \
   typedef TYPE vjSingletonBase;                \
   typedef vjSingletonPtr vjSingletonConstructor();  \
   static vjSingletonConstructor *sInstanceConstructor; \
public:                                         \
   static TYPE* instance( void )

#define vjSingletonImp( TYPE )                  \
   TYPE::vjSingletonConstructor *TYPE::sInstanceConstructor=TYPE::constructor; \
   TYPE::vjSingletonPtr TYPE::constructor( void )    \
   {  return new TYPE; }                        \
   TYPE* TYPE::instance( void )                 \
   {                                            \
      static vjMutex singleton_lock1;           \
      static TYPE* the_instance1 = NULL;        \
                                                \
      if (the_instance1 == NULL)                \
      {                                         \
         vjGuard<vjMutex> guard( singleton_lock1 );  \
         if (the_instance1 == NULL)             \
         /*{ the_instance1 = new TYPE; }*/      \
         { the_instance1 = sInstanceConstructor(); } \
      }                                         \
      return the_instance1;                     \
   }

#define vjAbstractSingletonImp( TYPE )          \
   TYPE::vjSingletonConstructor *TYPE::sInstanceConstructor=NULL; \
   TYPE* TYPE::instance( void )                 \
   {                                            \
      static vjMutex singleton_lock1;           \
      static TYPE* the_instance1 = NULL;        \
                                                \
      if (the_instance1 == NULL)                \
      {                                         \
         vjGuard<vjMutex> guard( singleton_lock1 );  \
         if (the_instance1 == NULL)             \
         /*{ the_instance1 = new TYPE; }*/      \
         { the_instance1 = sInstanceConstructor(); } \
      }                                         \
      return the_instance1;                     \
   }

#define vjDerivedSingletonImp( TYPE )           \
   TYPE::vjSingletonPtr TYPE::constructor( void )    \
   {  return new TYPE; }                        \
   bool TYPE::registerSingleton()               \
   {                                            \
     printf("registering singleton " #TYPE "\n");    \
     sInstanceConstructor=TYPE::constructor;    \
     return true;                               \
   }                                            \
   bool TYPE::isRegistered=TYPE::registerSingleton(); \
   TYPE* TYPE::instance( void )                 \
   {                                            \
      return static_cast<TYPE*>(vjSingletonBase::instance()); \
   }
```

4.1.5 Swaplock Support

For a proper display synchronization, all nodes should synchronize to swap their frame buffers (swaplock). VR Juggler does not include any swaplock support. It assumes that the underlying system is responsible for swapping synchronization. This is for example the case on an SGI Onyx system. Net Juggler is aimed at running VR applications on machines built of commodity components that usually do not support swaplock. So Net Juggler includes a software swaplock support.

VR Juggler rendering occurs as follow:

```
drawmanager->draw(); // start drawing
drawmanager->sync(); // wait until frame is displayed on
                     // screen
```

For swaplocking we use a synchronization barrier that forces the different nodes to wait each other before to swap their frame buffers. This synchronization barrier is preceded by a call to `swapReady` and followed by a call to `swap`, two methods that were added to the `drawmanager` class. The sequence of calls in the `NetKernel` main loop is the following:

```
drawmanager->draw();      // start drawing
drawmanager->swapReady(); // wait until rendering is finished
// and frame is ready to be displayed
netapi->barrier();        // synchronization with other nodes
                          // (swaplock)
drawmanager->swap();      // display frame on screen
drawmanager->sync();      // wait until frame is displayed on
                          // screen
```

For OpenGL, `swapReady()` is based on a call to `glFinish()`.

Swaplock for Performer is not yet supported.

4.2 Communication Library

Because we assume the communication library used may not be thread safe, calls to the NetAPI are all performed by the same thread (the kernel thread).

4.2.1 MPI

Thread Safe

MPI implementations are not necessarily thread safe.

Collective Communication Implementation

Depending on your MPI implementation, collective operations may not be optimized for Net Juggler communication requirements. For example the allgather operation is typically implemented by having all processors shifting messages in a ring. This is efficient for large messages, but for small messages a gather followed by a broadcast is generally more efficient.

The `NetAPI_MPI` class contains constants that are used to select between different implementations (see `NetAPI_MPI/NetAPI_MPI.h`):

- `NETAPI_MPI_BROADCAST`: If set to 1 the `NetAPI_MPI:allGather` method is implemented with the MPI `mpi_bcast` function.
- `NETAPI_MPI_BARRIER`: If set to 1 the `NetAPI_MPI:allGather` method is implemented with the MPI `mpi_barrier` function.
- `NETAPI_MPI_ALLGATHER`: If set to 1 the `NetAPI_MPI:allGather` method is implemented with the MPI `mpi_allgather` function.

By default all theses constant are set to 1. Refer to the code of the `NetAPI_MPI` class to know the other implementations available. By changing the constant values you can select different implementations. The `netapi_mpi_test` program can be used to measure performances.

Appendix A

Genlock and Active Stereo

A.1 Introduction

High quality image projections and active stereo rendering in multi-display environments require genlocking video signals, i.e. a video retrace synchronization. Except for some high end cards, off-the-shelf graphics cards do not support genlock. This is an important limitation for virtual reality PC clusters.

We describe in this section the SoftGenLock library. It provides software support for multi-display genlocking. The library also enables quad buffer page flipped stereo that today's graphics card drivers generally do not support.

The association of Net Juggler and SoftGenLock makes possible to run a virtual reality application with active stereo and multi displays on a PC cluster.

For the moment, SoftGenLock is developed only for Linux and supports NVIDIA graphics cards.

A.2 Genlocking Video Signals

A.2.1 Algorithm

When a node detects the vertical retrace for its video cards, it starts a synchronization barrier and measures the time taken to proceed this barrier. If the delay is considered too long the machine slows down its video retraces. When the delay is considered small enough video retrace goes back to its normal speed. The algorithm is described bellow:

```
wait = false
Loop:
   Wait for vertical retrace
   Barrier
   t = barrier execution time
   If (t > time to retrace one image) then
      display (``IMAGE LOST'')
   else
      if (t > too_long   and  wait == false) then
          Slow down video signal
          wait = true
      end
      if (t < small_enough   and wait == true) then
          Go back to normal signal speed
          wait = false
      end
   end
end
```

A.2.2 Setting Time Parameters

The algorithm requires the following data:

- The time `sync_time` required to execute a barrier when all barrier calls are synchronized.

- The extra time `delay` introduced during one image retrace when the signal is slowed down.

The highest quality genlock is achieved by setting the variables `small_enough` to `sync_time` and `too_long` to `sync_time+delay`.

Active stereo does not require a perfect genlock. The signals should stay synchronized within a range that ensures the shutters flip without an eye see the wrong image. This range depends on the shutter latency and the time the order to flip shutters is sent.

A.2.3 Synchronization Barrier

The performance of the synchronization barrier is very important as it establishes the genlock quality. Video signals can not be synchronized with a precision inferior to `sync_time`.

We obtained good results with a Myrinet Network were 4 PCs can be synchronized in about 30 microseconds. We also developed a dedicated synchronization network for swaplock and genlock based on a TTL_PAPERS network (see `www.aggregate.org`). The synchronization requires less than 5 microseconds allowing a very precise genlock.

A.2.4 Video Signal Access

To implement this algorithm requires to know how to detect the vertical retrace (VR) and how to slow down the video signal. There is no standard functions.

Several approaches were considered:

- The X Synchronize extension. It provides timers synchronized with the retrace. Unfortunately XFree86 implementation do not support these timers.

- The XF86DGA implementation. It allows to modify the screen area displayed in synchronization with the retrace. It could be used to detect the vertical retrace and to select the image displayed (for stereo). But this extension appeared not to be compatible with OpenGL programs and do not allow to slow down the signal.

- The XF86VidMode extension. It allows to modify display parameters. But this extension do not permit to detect the vertical retrace. Moreover, the NVIDIA driver implementation ignores the orders to slow down the signal. A modification of the parameters initializes the video signal and restarts it with delay we were not able to control.

- The XFree86 server and the associated graphics card driver. NVIDIA driver source codes are not available.

- Hacking the graphics card registers. Most graphics cards are VGA compatible, having a status register and CRTC registers. The status register can be used to detect the vertical retrace. The CRTC registers can be used to modify the video signal. However, we experienced register corruption problems when accessing the registers concurrently with the drivers.

SoftGenLock is based on the last approach. By accessing registers just before the vertical retrace, register corruption almost never occur. A more reliable solution for the issue is also currently studied.

A.2.5 Vertical Retrace Waiting

Because we do not have access to the vertical retrace interrupt, other approaches to detect and wait for the vertical retrace must be considered.

We use the graphics card status register to detect the vertical retrace. We can poll the state of this register, but this active waiting is CPU time consuming. To free the CPU, we use a real-time timer that is started after each vertical retrace. It is set up to wake up the SoftGenLock thread just before the next vertical retrace.

We use the RT-Linux system (see `www.rtlinux.org`). The high precision of RT-Linux timers permits to reduce the active waiting to a few tens of microseconds for each retrace.

Note that this overhead could be reduce by dynamically refining the SoftGenLock thread sleeping time.

A.3 Active Stereo Support

Active stereo display requires the graphics card to compute two different images, one for each eye, and to display them alternatively switching at each video retrace. Shutter glasses are synchronized with the retrace signal to ensure each eye only sees its image.

XFree86 is set up to have a virtual buffer twice as high as the the screen display. The 3D software (NET/VR Juggler) must then be set up to write the left eye image in the top half buffer and the right eye image in the down half buffer. Next, each time a vertical retrace is detected the displayed part of the buffer is changed and a signal is sent to the shutter glasses.

To alternate the image displayed, SoftGenLock modifies the display start address in the CRTC registers.

To send the stereo sync signal to shutter glasses we tested two solutions, depending on how shutter glasses are connected to the PC:

- The signal is written to a parallel port register. Generally shutter glasses are not connected to the parallel port but it is easy to brew a home made adaptor.
- The stereo signal is written to the DDC SDA pin of the SVGA video port. For NVIDIA cards the DDC bit is set by writing into the CRTC register `0x3f`. A stereo enabler adaptor is then necessary to extract the signal and forward it to the glasses. Using the ELSA Revelator stereo enabler adaptor allows to connect any glasses that has a VESA standard 3-pin mini-DIN stereo connector.

Once calls for stereo display are added to SoftGenLock the main algorithm becomes the following:

```
wait = false
frame = 0
Loop:
   Wait for vertical retrace
   Barrier
   t = barrier execution time
   set_display_starting address( (frame \% 2) *  image_size)
   set_stereo_sync_signal (( frame \% 2 ) ? right_eye : left_eye)
   frame = frame + 1
   If (t > time to retrace one image) then
      display (``IMAGE LOST'')
   else
      if (t > too_long   and  wait == false) then
          Slow down video signal
          wait = true
      end
      if (t < small_enough   and wait == true) then
          Go back to normal signal speed
          wait = false
      end
```

```
  end
end
```