INTRO

Babe

A cross-platform framework for realtime system

A challenge : Making system development an “**almost easy**” **task**.

NINA

Nomad Interface designed for Networked Application

A challenge : **Systematic reuse** of networked objects

Babel

A real life example of use of **Babe** and **NINA**

And how those frameworks improved the **ease of development**

Babe

Babe : Architecture Overview

3 Layered Architecture

Framework

Plugins

Application

3 Layered Architecture

Framework	Plugins	Application
libBabe.so	libBabe_NinaNetwork.so libBabe_PortAudioAudio.so libBabe_QtGui.so libBabe_SpeexCodec.so	Babel

Gamut of reusability

99 %

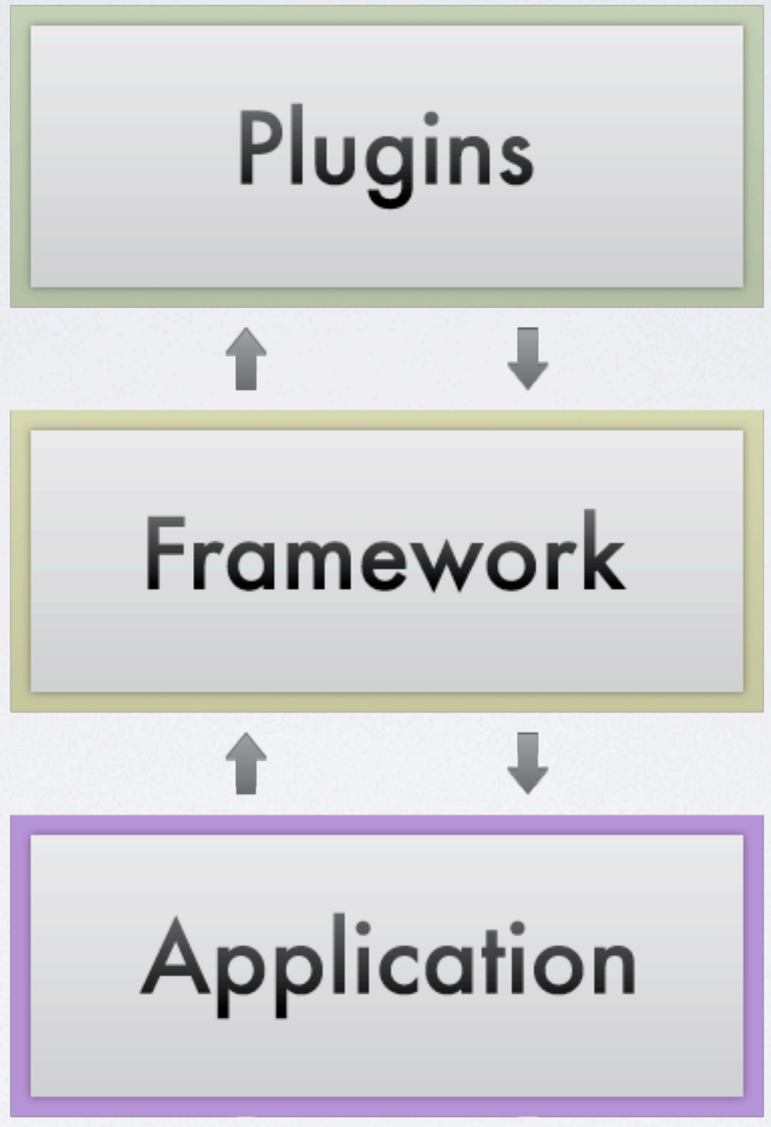
50 %

1%

Framework

Plugins

Application


Plugins
Framework
Application

Babe : System and Plugin Management

A realtime system is composed out of **many subsystems**.

E.g. : GUI, Network, Audio, Input, Command...

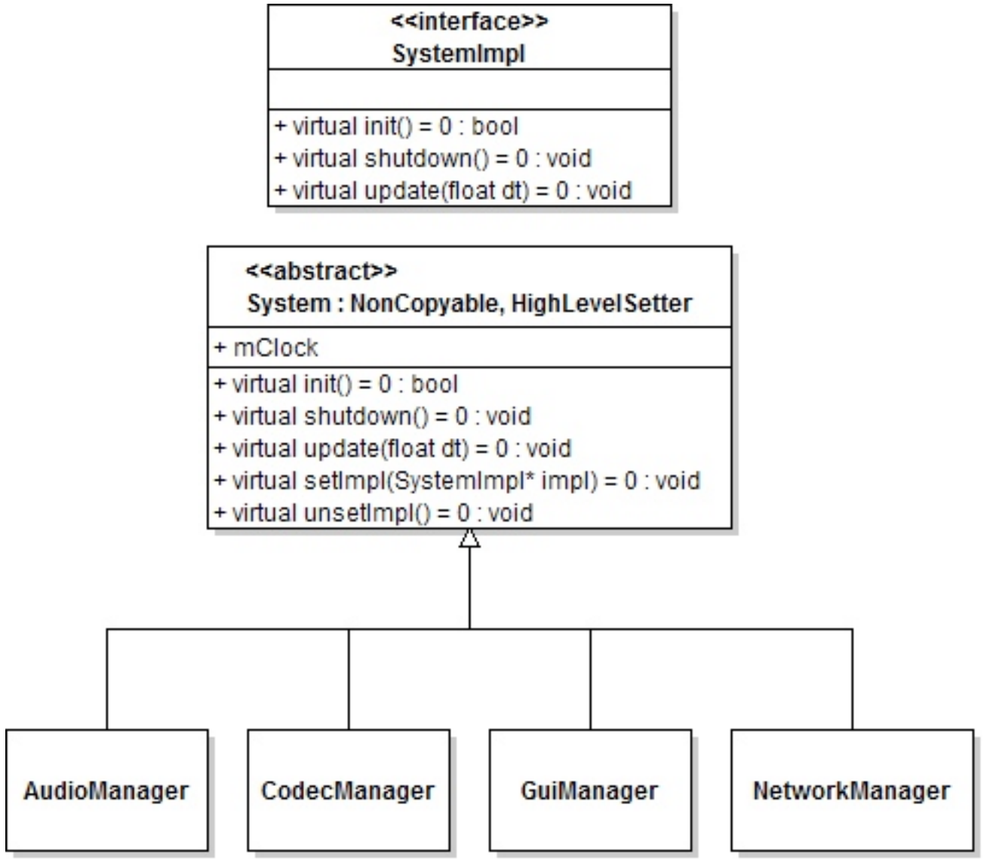
Each of these systems has to be **instantiated**, **initialized**, **updated** and **shutdown**.

For so many reasons, system development can be a **difficult task**.

The main goal of Babe is to **ease up** system development.

With the way it is designed, Babe makes you use **design patterns** without having you know about it.

In Babe, Systems are implemented in a form of a **bridge pattern**.



The actual implementation is a **plugin.**

If the load of the plugin fails, there is always a **dummy implementation** (Null pattern) to fall back to.

For example, you can start without any system implementations and **load them one at a time**.

```
> loadplugin Babe_QtGui
```

In Babe, Systems are **observers** (pattern) that wait to be notified for initialization, update and shutdown.

Babe has **built-in systems**, but if you are not happy with them, you can always **write your own**.

You write your own **systems** as a **plugin.**

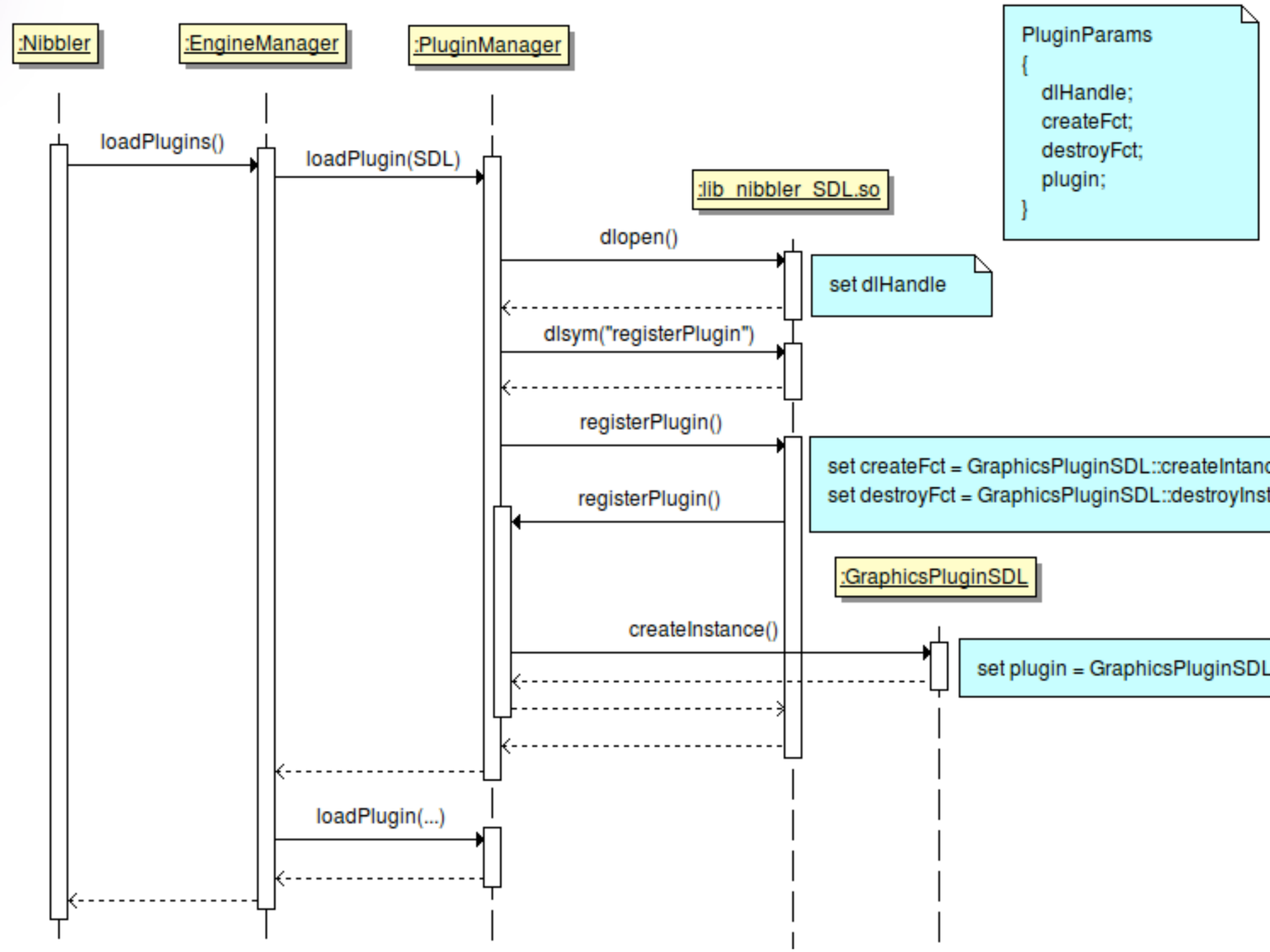
Babe has a built-in **System Manager** and **Plugin Manager** so you don't have to worry about this, Babe does all of the management for you.

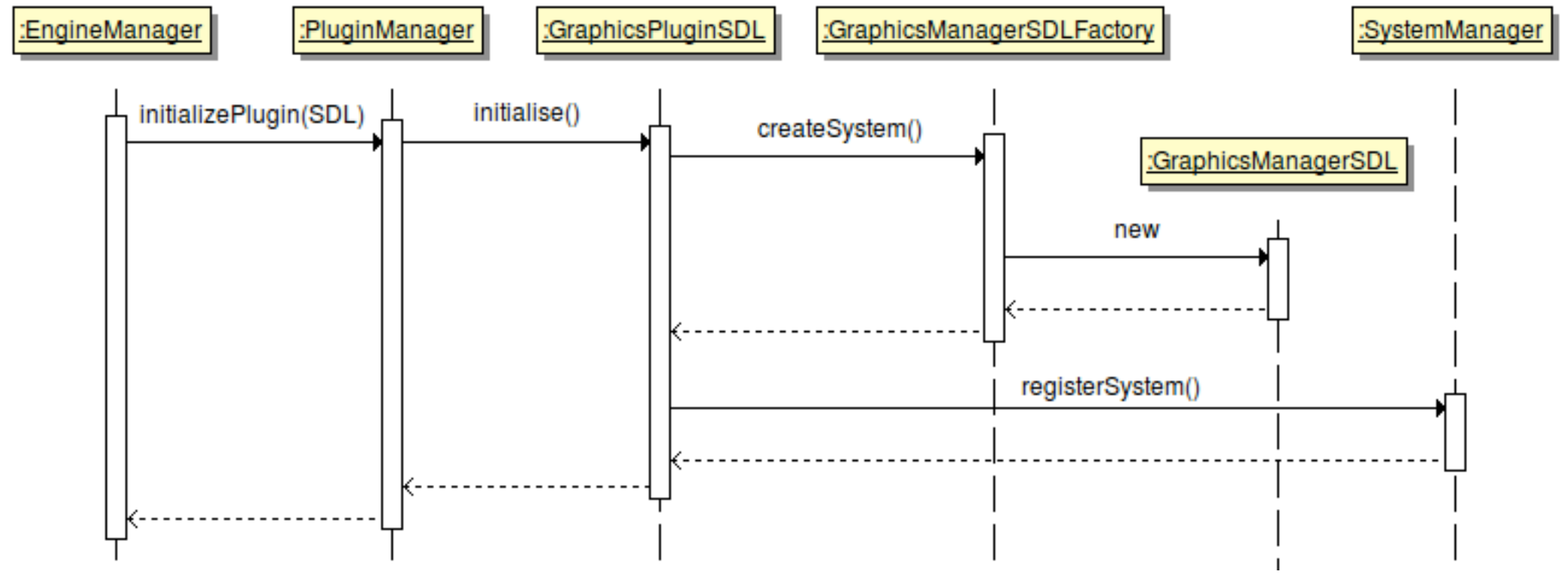
Write your systems as if they were in the same **compilation unit**.

Making them a plugin is as simple as **10 lines of code** that you don't even have to come up with.

```
typedef Babe::PluginTpl<AudioManagerPortAudioImpl> AudioPluginPortAudio;

extern "C"
{
  BABE_DLLREQ bool registerPlugin(Babe::PluginManagerImpl* pluginManager, void* dlHandle)
  {
    Babe::PluginManagerImpl::PluginParams pp;
    pp.dlHandle = dlHandle;
    pp.createFct = &AudioPluginPortAudio::createInstance;
    pp.destroyFct = &AudioPluginPortAudio::destroyInstance;
    pp.systemName = "AudioSystem";
    pp.pluginName = "AudioPluginPortAudio";
    pp.version = "1.0";
    if (!pluginManager->registerPlugin(&pp))
      return false;
    return true;
  }
}
```


:Nibbler
:EngineManager
:PluginManager
:lib_nibbler_SDL.so
loadPlugins()
loadPlugin(SDL)
dlopen()
set dlHandle
dlsym("registerPlugin")
registerPlugin()
registerPlugin()
set createFct = GraphicsPluginSDL::createIntance
set destroyFct = GraphicsPluginSDL::destroyInstance
:GraphicsPluginSDL
createInstance()
set plugin = GraphicsPluginSDL
loadPlugin(...)
PluginParams
{
dlHandle;
createFct;
destroyFct;
plugin;
}


:EngineManager
:PluginManager
:GraphicsPluginSDL
:GraphicsManagerSDLFactory
:SystemManager
initializePlugin(SDL)
initialise()
createSystem()
:GraphicsManagerSDL
new
registerSystem()

The **abstractions** of the "ready to be developed" systems are as **simple** as possible to keep **interdependencies** as **small** as possible.

Once you are done developing your system and ready to test it,
There is a **script** to describe in what **order** you want them to be **instantiated**, **initialized**, **updated** and **shutdown**.
You can even specify the **target rate**.

A script to **describe the systems** you want to use

```
<?xml version="1.0"?>
<config>
  <systems>
    <system type="Audio"   plugin_path="Babe_PortAudioAudio" plugin_name="AudioPluginPortAudio"/>
    <system type="Codec"   plugin_path="Babe_SpeexCodec"     plugin_name="CodecPluginSpeex"/>
    <system type="Gui"     plugin_path="Babe_QtGui"          plugin_name="GuiPluginQt"/>
    <system type="Network" plugin_path="Babe_NinaNetwork"    plugin_name="NetworkPluginNina"/>
  </systems>
  <network host="10.224.7.39" port="sip" family="IPv4"/>
</config>
```

Babe : a Command Driven Communication

System interdependencies is a real **pain in the ass**

System **interdependencies** can happen **anywhere** : at the instantiation of a system, the initialization, the update, the shutdown...

E.g. : The GUI system is done, but the Network system is not. So you are unable to test the GUI. The team in charge of the GUI has to wait for the team in charge of the Network.

Encapsulating systems interaction in **commands** (pattern) gives us a way to **simulate** other systems.

Simply open the console and type in your **command**.

Et voila, a system action has been **simulated**.

```
> userstatus Nina connected
```

A way to encapsulate the **application logic**.

Babel.cpp :

```
Babel::Babel()
  : mApp(new Babe::ApplicationManager("Babel")),
    mCmdMgr(Babe::CommandManager::getSingletonPtr())
{
  registerCommands();
}

void    Babel::run()
{
  mApp->run();
}
```

But most importantly a way to **develop** systems **independently** from other systems.

NINA

THANKS

First of all we would like to specially thanks Douglas C.Schmidt, W.Richard Stevens and all their coworkers for their amazing jobs around the software engineering and the network programming.

OVERVIEW

NINA has been released under ISC license.
Inspired by the ADAPTIVE Communication Environment Framework (ACE).

Dynamic library.

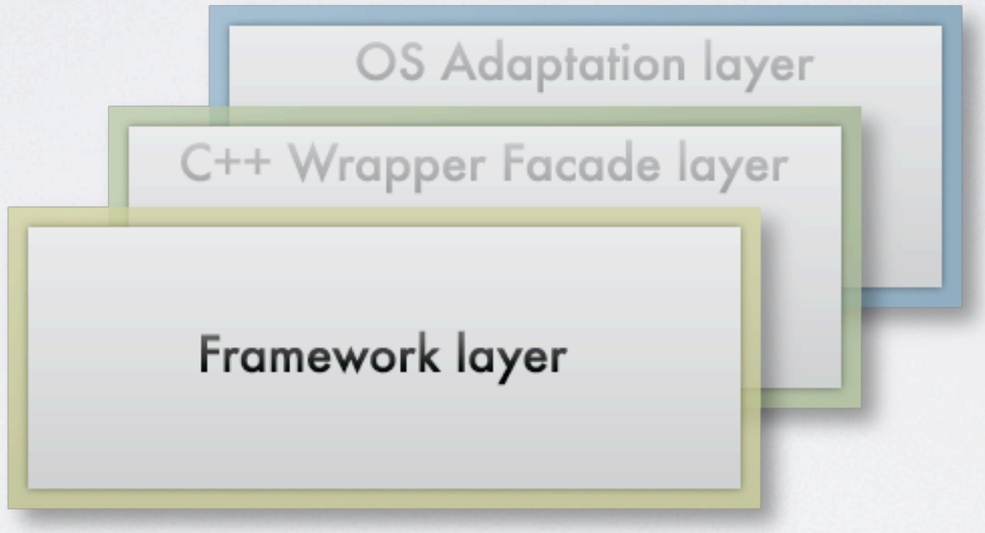
Cross platform (*BSD, Linux, Microsoft Windows)

Native IPv4/IPv6 library

~15 000 lines of code

Fully documented at http://nina.exxo.us

LAYERS


OS Adaptation layer
C++ Wrapper Facade layer
Framework layer

Organized in layers, abstracting system dependent operations.
Uses of advanced design patterns to ensure genericity.

PACKET CLASS RELATIONSHIPS

Wraps up data across network and different plateforms
Marshals/Demarshals with memory alignement and byte order
in mind Fragments a stream into packets.

ACCEPTATION CLASS RELATIONSHIPS

Accepts new connections and spawns appropriate services. Services process indication events received and dispatch by a reactor.

EVENT HANDLING CLASS RELATIONSHIPS

Common way to associate an event with a handle.
Perform specific services in response to particular events.

DISPATCHING CLASS RELATIONSHIPS

Use of Reactor pattern to dispatch events.

Associate an event handler with an event.

Supported synchronous event demultiplexer : Poll, Epoll, Select, Kqueue, WFMO.

AN ECHO-REPLY SERVER WITH THE NINA API

```
typedef NINA::SelectPolicy POLICY;

int main(void)
{
   NINA::Reactor<POLICY>                                  reactor;
   NINA::InetAddr                                         local_address;
   NINA::Acceptor<EchoReply, NINA::SockAcceptor, POLICY>  acceptor(&reactor);

   NINA::Init  startup;

   local_address.wildcardQuery("echo", "tcp");
   acceptor.open(local_address, true);
   reactor.handleEventsLoop();
   return 0;
}
```


Babel

Babel

An Overview of the Systems developed for Babel And their Abstraction

Four systems were developed for Babel, a **GUI** system, a **Network** system , an **Audio** system and a **Codec** system.

The abstraction of each systems is very **simple**.
It helps keep things **clean** and **reduces interdependencies**
between systems.

The **GUI** System has only one abstract function.

```
void notify(std::string const& notification);
```

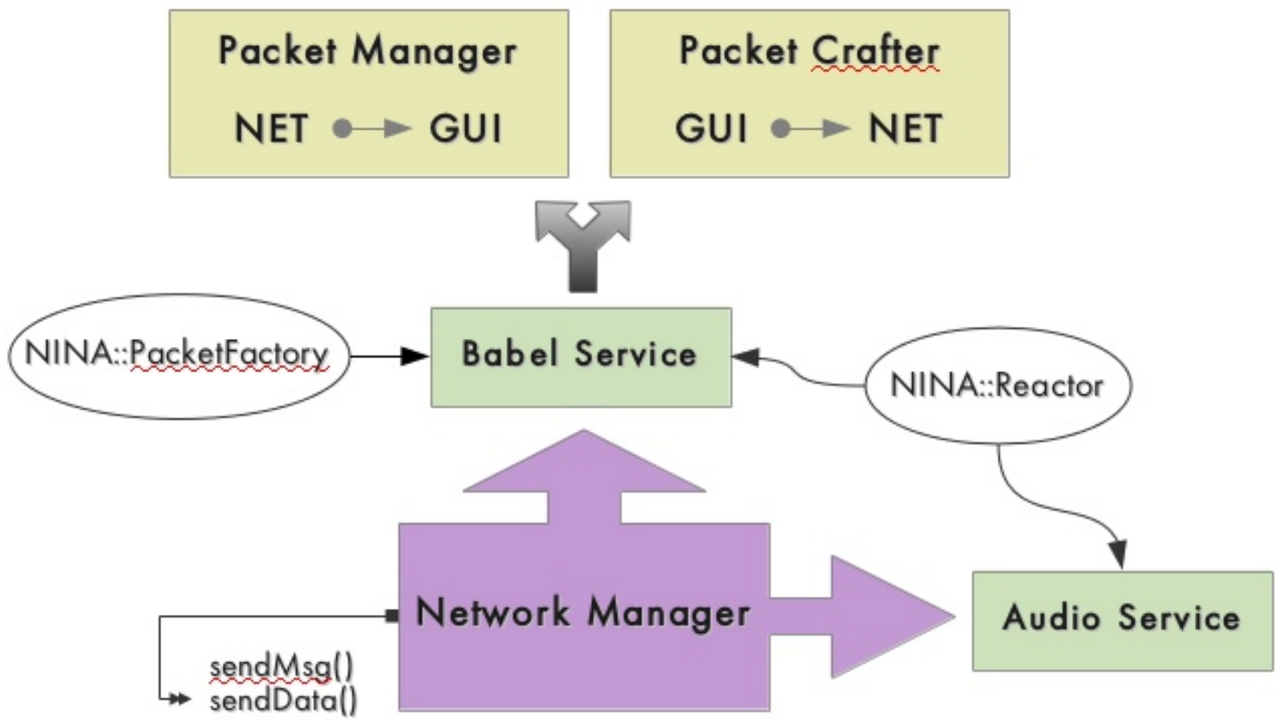
The **Network** System has four abstract functions.

```
bool isConnected() const;
bool connect(std::string const& hostName, std::string const& port,
             NetworkManagerImpl::eConnectionFamily family = NetworkManagerImpl::IPV4);
bool sendMessage(std::string const& message);
bool sendData(size_t dataSize, void* data);
```

The **Network** System has four abstract functions.

```
bool isConnected() const;
bool connect(std::string const& hostName, std::string const& port,
             NetworkManagerImpl::eConnectionFamily family = NetworkManagerImpl::IPV4);
bool sendMessage(std::string const& message);
bool sendData(size_t dataSize, void* data);
```

Network plugin architecture



The **Audio** System has four abstract functions.

```
void on();
void off();
AudioFrame* recordAudioFrame();
void playAudioFrame(AudioFrame* frame);
```

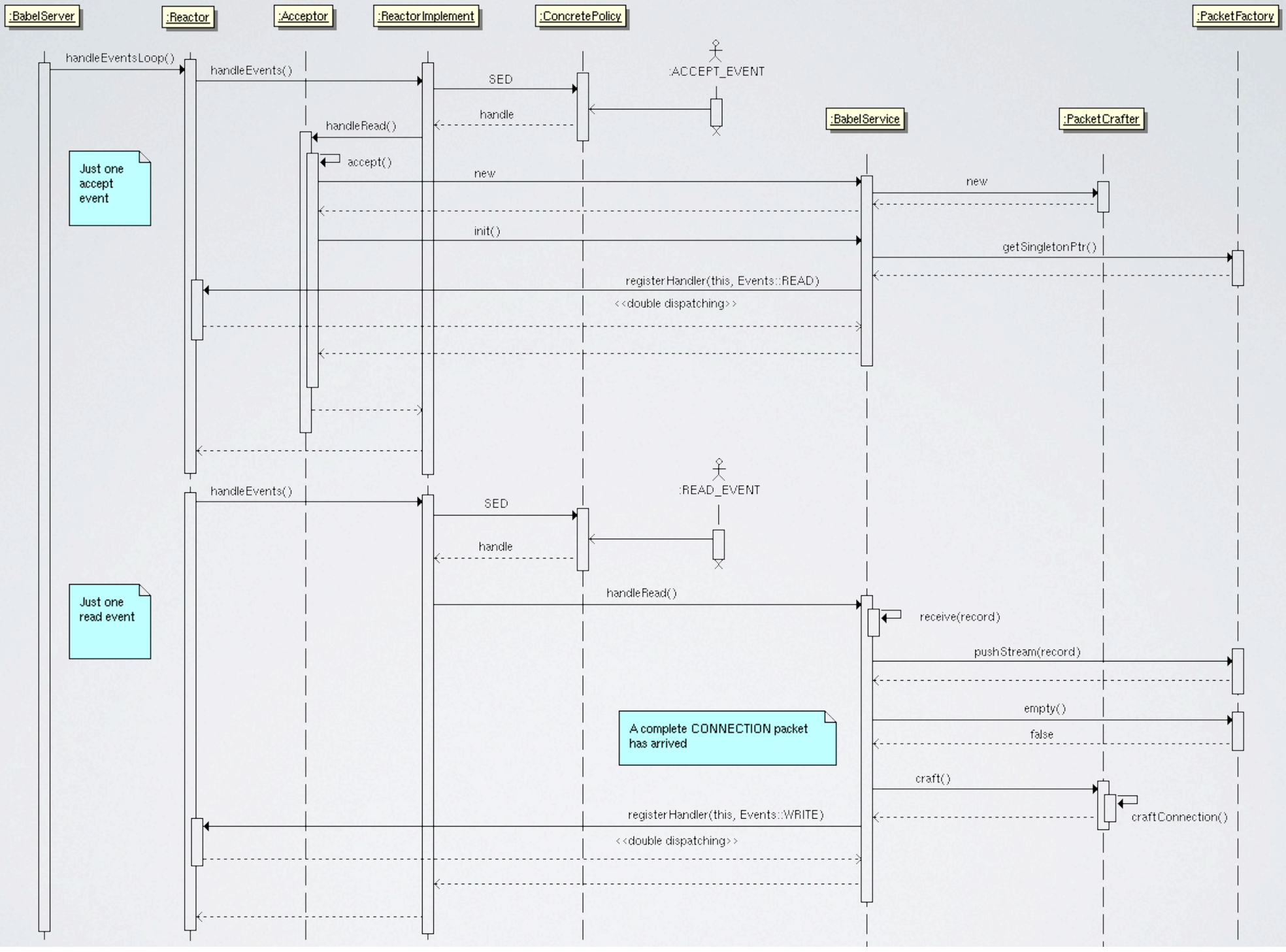
The **Codec** System has two abstract functions.

```
void encode(AudioFrame& frame);
void decode(AudioFrame& frame);
```

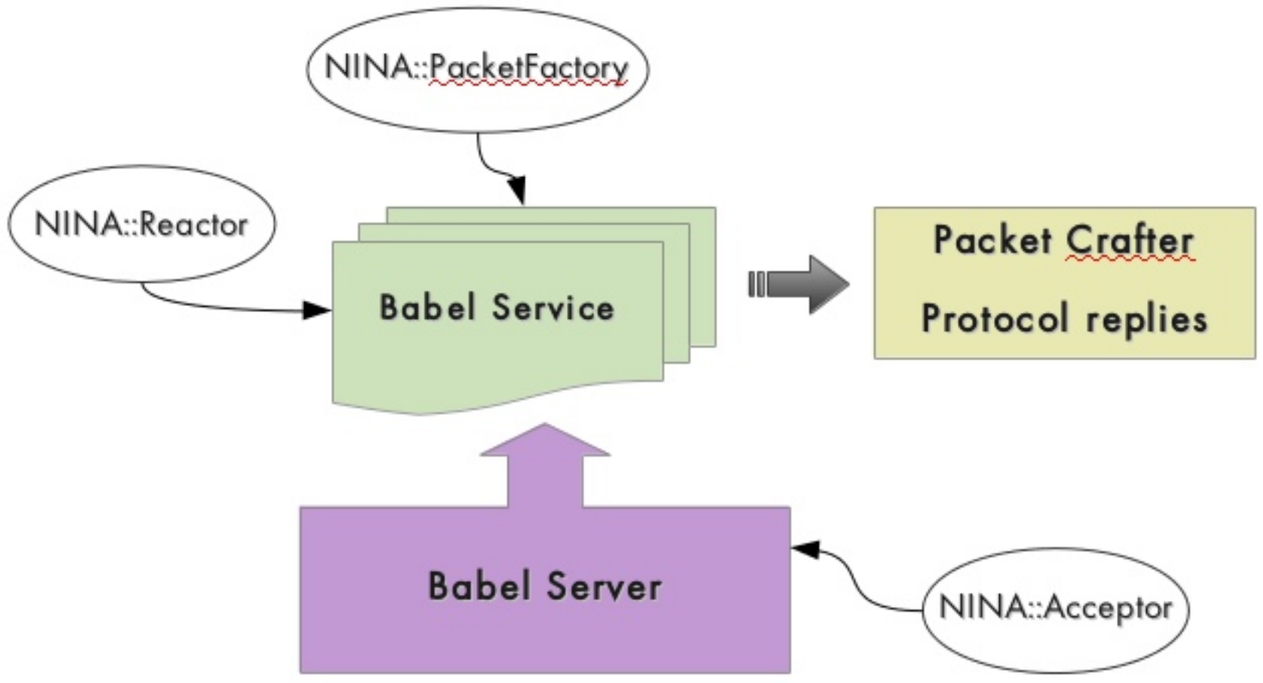
Babel

An Overview of the Server for Babel And its Abstraction

Example of a logging session.


:BabelServer
:Reactor
:Acceptor
:ReactorImplement
:ConcretePolicy
:PacketFactory
handleEventsLoop()
handleEvents()
SED
:ACCEPT_EVENT
handle
:BabelService
:PacketCrafter
handleRead()
accept()
Just one accept event
new
new
init()
getSingletonPtr()
registerHandler(this, Events::READ)
<<double dispatching>>
handleEvents()
SED
:READ_EVENT
handle
handleRead()
Just one read event
receive(record)
pushStream(record)
empty()
false
A complete CONNECTION packet has arrived
craft()
craftConnection()
registerHandler(this, Events::WRITE)
<<double dispatching>>

Server architecture



Babel

An Overview of the commands that implement the Application Logic

With Babe, **writing** and **registering** commands is a **simple task**.

```
mCmdMgr->registerCommand("userconnect",
                         new Babe::CommandFromStringFactoryTpl<UserConnectCmd>,
                         "userconnect userName userPassword\n man");
```

The Application logic of Babel is encapsulated in **28 commands**.

There is commands for :

- User connection management
- User status management
- Contact management
- Call management
- Chat management
- Error management

User connection management

userconnect *userName userPassword*

Command to be executed from the GUI to inform the Network of a new user connection.

User connection management

userconnectionsucceeded

Command to be executed from the Network to inform the GUI the connection succeeded.

User status management

userstatus *login status*

Command to be executed from the Network to inform the GUI about a user status.

User status management

updateuserstatus *status*

Command to be executed from the GUI to inform the Network the connected user changed his status.

Contact management

addcontact *login*

Command to be executed from the GUI to inform the Network the connected user wants to add a new contact.

Contact management

contactrequest *login*

Command to be executed from the network to inform the gui a user wants to add the connected user as a new contact.

Call management

requestcall *login*

Command to be executed from the GUI to inform the Network the connected user wants to call a contact.

Call management

callrequest *login*

Command to be executed from the Network to inform the GUI a contact wants to call the connected user.

Chat management

sendtext *login message*

Command to be executed from the GUI to send a message to the specified login.

Error management

connectionlost

Command to be executed from the Network to inform the GUI the connected user received a message from login.

To have an **exhaustive list** of the commands
along with their man, simply type

ctrl + shift + F2 ,

then open up the console and type in the desired command.

```
> userstatus Nina connected
```

And now the **bonus** you were all waiting for...