echo Documentation

Release 1.0

ViCoS

February 11, 2016

Contents

Contents:

CHAPTER 1

Introduction to Echo

Echo is a inter-process communication library for C++ and Python focused on performance and ease of use. It provides a publish/subscribe model for message passing and a simple data marshalling system with minimal overhead. If advanced data marshalling is required, various third party solutions (such as Flat Buffers) are supported.

Installation and use of the system are designed to be as simple as possible, and installation requires minimal dependencies. To get started with installing the system, click here. If you've already installed the system, read how to start using it here (c++) or here (python).

CHAPTER 2

Installation

2.1 Supported platforms

Currently, Echo only supports Linux platforms. Support for other POSIX based platforms is planned

2.2 Dependencies

The core Echo system with bindings for c++ requires no additional dependencies. If you want to use the included Python bindings, PyBind11 is required. Since one of the provided examples shows video streaming with the ose of the OpenCV platform, OpenCV is required to build this example. Additionally, you will need CMake to build the project.

2.3 Installation

First, create a new directory and download the project from Bitbucket:

```
mkdir project_directory_path
cd project_directory_path
git clone https://bitbucket.org/UrbanSk/echolib
cd echolib
```

Then, use CMake to build the project:

```
cmake .
make
```

2.4 Running the examples

Echo comes with several example programs that demonstrate it's use. After building the project, they should be available in the ./bin directory. The following files should be created:

```
echodaemon
chat (simple chat application)
chunked (a demonstration of splitting big messages into smaller, more optimal chunks)
opencv (a demonstartion of sharing video using opencv)
```

Echo is based on a central process or daemon, which runs in the background and manages the communication between processes. Because of this, the daemon process must always be running in the background when you wish to use Echo. When starting the daemon process, you must prowide a socket name that other proccesses will connect to. To start the daemon, simply run:

```
./echodaemon path_to_socket/socket_name
```

Alternativelly, if you wish to communicate over IP sockets, run the daemon with:

```
./echodaemon -i port
```

With the daemon running, you can now start programs that will communicate with each other. For example, starting two instances of the chat example will allow you to send text messages between the instances.

Echo is now ready for use. Click `here` to start using the system.

CHAPTER 3

Getting started - Python

3.1 Basic concepts

- Echo is an interprocess communication library. It allows different processes or programs to share data between themselves.
- Echo uses a publish/subscribe system to allow for organized sharing of data between processes. All communication is organized through named channels. A message isn't sent to specific processes, but to a channel, and is then distributed automatically to every process that is subscribed to that channel.
- Handling of received messages is done through callback functions. When a process subscribes to a channel, it also provides a callback function that will be called on the received messages.
- Echo uses a central process (also called a dameon) to manage it's communication. This the deamon needs to run in the background if you want to use the library. To start the daemon, navigate to the build directory of the project and run ./echodaemon socket_path, where socket path is the path to a socket through which the processes will communicate. The programs you are writing will use this socket path to connect with the daemon.
- All Echo methods are thread safe.

3.2 Writing a simple program

Let's examine the included chat example to get an idea of how to write programs using Echo. A chat application is one of the simplest examples of interprocess communication.:

```
import sys
import time
import pyecho
import thread


def message(reader):
    name = reader.readString()
    message = reader.readString()
    print name + ": " + message

def write(communicator,pub,name):
    while communicator.isConnected():
        message = raw_input()
        writer = pyecho.MessageWriter()
        writer.writeString(name)
```

```
        writer.writeString(message)
        pub.send(writer)

def main():
    if len(sys.argv) < 2:
        raise Exception('Missing socket')

    communicator = pyecho.Communicator(sys.argv[1])

    name = raw_input("Please enter your name:")

    sub = pyecho.Subscriber(communicator, "chat", "string pair", message)
    pub = pyecho.Publisher(communicator, "chat", "string pair")

    thread.start_new_thread(write,(communicator,pub,name,))
    try:
        while communicator.wait(10):
            time.sleep(0.001)
    except KeyboardInterrupt:
        communicator.disconnect()

    sys.exit(1)

if __name__ == '__main__':
    main()
```

All of Echo's functions are defined in the pyecho library, so we first need to include it:

```
import pyecho
```

If we want to use the library, we first need to establish a connection with the central daemon process. This is done by creating a communicator object to which we pass the socket path that we specified when running the daemon. In this case, the socket path is provided to the program as an argument:

```
communicator = pyecho.Communicator(sys.argv[1])
```

Next, we create a publisher and a subscriber. The publisher will send the messages that the user enters, while the subscriber will display received messages on screen:

```
pub = pyecho.Publisher(communicator, "chat", "string pair")
```

When creating the publisher, we need to specify the communicator we defined earlier, a channel name (in our case chat) and the type of a message. Our messages will consist of two strings (one for the name of the user and one for the user's message), so the type is "string pair":

```
sub = pyecho.Subscriber(communicator, "chat", "string pair", message)
```

Creating a subscriber is almost the same, except we also need to provide a callback function. In this case, the function in called message and is defined on top of the source code:

```
def message(reader):
    name = reader.readString()
    message = reader.readString()
    print name + ": " + message
```

Everytime we receive a message, this callback finction will be called on it. Extracting data from the message is as simple as calling appropriate read methods on the recieved message. Since our chat messages consist of a pair of strings, we call readString() twice, then print the result. Now we know how to handle received messages. Sending them works mostly the same:

```
def write(communicator,pub,name):
    while communicator.isConnected():
        message = raw_input()
        writer = pyecho.MessageWriter()
        writer.writeString(name)
        writer.writeString(message)
        pub.send(writer)
```

the write function represents the main loop of our chat program. It reads in a text message, then sends the user's name and the read message to all chat programs that are currently running (or, more specifically, to the chat channel on the daemon, which then handles the distribution). We first create a message writer, write the strings with writeString(), and send the message through the publisher we defined earlier. Now we simply start this function on a new thread:

```
thread.start_new_thread(write,(communicator,pub,name,))
```

The final thing left to do is define when we want to process the received messages. Since this is a simple chat program, we will simply continuously process them in the main thread:

```
while communicator.wait(10):
    time.sleep(0.001)
```

Every time we call communicator.wait(), the communicator will collect all messages send to the subscribers that were defined with it and call their callback function. Since we don't wand to completelly overload the communicator, we add a small delay into the loop. Once we are done, we need to safelly disconnect all publishers and subscribers from the daemon. We van do this using the disconnect() method of the communicator:

```
communicator.disconnect()
```

CHAPTER 4

Getting started - C++

4.1 Basic concepts

- Echo is an interprocess communication library. It allows different processes or programs to share data between themselves.
- Echo uses a publish/subscribe system to allow for organized sharing of data between processes. All communication is organized through named channels. A message isn't sent to specific processes, but to a channel, and is then distributed automatically to every process that is subscribed to that channel.
- Handling of received messages is done through callback functions. When a process subscribes to a channel, it also provides a callback function that will be called on the received messages.
- Echo uses a central process (also called a dameon) to manage it's communication. This the deamon needs to run in the background if you want to use the library. To start the daemon, navigate to the build directory of the project and run ./echodaemon socket_path, where socket path is the path to a socket through which the processes will communicate. The programs you are writing will use this socket path to connect with the daemon.
- All Echo methods are thread safe.
- Echo nativly uses recent C++11 and C++14 features such as smart pointers and anonymous functions. This means that you don't need to use libraries like boost for this functionallity.

4.2 Writing a simple program

Let's examine the included chat example to get an idea of how to write programs using Echo. A chat application is one of the simplest examples of interprocess communication.:

```
#include "client.h"
#include "datatypes.h"

template<> inline shared_ptr<Message>
echolib::Message::pack<pair<string, string> >
(int channel, const pair<string, string> &data)
{
    MessageWriter writer(data.first.size() + data.second.size() + 8);

    writer.write_string(data.first);
    writer.write_string(data.second);

    return make_shared<BufferedMessage>(channel, writer);
```

```
}

template<> inline shared_ptr<pair<string, string> >
echolib::Message::unpack<pair<string, string> >(SharedMessage message)
{
    MessageReader reader(message);

    string user = reader.read_string();
    string text = reader.read_string();

    return make_shared<pair<string, string> >(user, text);
}

int main(int argc, char** argv)
{

    SharedClient client = make_shared<echolib::Client>(string(argv[1]));

    string name;
    cout << "Enter your name\n";
    std::getline(std::cin, name);

    std::mutex mutex;

    function<void(shared_ptr<pair<string, string> >)> chat_callback =
    [&](shared_ptr<pair<string, string> > m){
        const string name = m->first;
        const string message = m->second;
        std::cout<< name << ": " << message << std::endl;
    };

    function<void(int)> subscribe_callback = [&](int m){
        std::cout << "Total subscribers: " << m << std::endl;
    };

    TypedSubscriber<pair<string, string> > sub(client, "chat", chat_callback);
    SubscriptionWatcher watch(client, "chat", subscribe_callback);
    TypedPublisher<pair<string, string> > pub(client, "chat");

    std::thread write([&](){
        while (client->is_connected()){
            string message;
            std::getline(std::cin, message);
            pub.send(pair<string, string>(name, message));
        }

    });

    while(client->wait(10)){
        std::this_thread::sleep_for(std::chrono::milliseconds(10));
    }

    exit(0);
}
```

Echo's functionallity is defined in several header files, so we need to include them first:

```
#include "client.h"
#include "datatypes.h"
```

First, we need to define the datatype we will be sending. In this example, we will be sending a pair of two strings - a pair<string,string>. If we want to transfer messages between two processes, we first need to tell echo how to convert them into an appropriate binary form. To do this, we need to define two functions, one to encode the object into a binary representation and one to decode the received object from it's binary representation. This is done through the first two function:

```
template<> inline shared_ptr<Message>
echolib::Message::pack<pair<string, string> >(int channel, const pair<string, string> &data)
{
    MessageWriter writer(data.first.size() + data.second.size() + 8);

    writer.write_string(data.first);
    writer.write_string(data.second);

    return make_shared<BufferedMessage>(channel, writer);
}
```

The first function is pack, which transforms the c++ object into the appropriate binary representation. It ovverrides the base pack method of the message class. The important parts of the function signature is the template argument of pack, which represents the datatype we will be sending, and the second argument, which is the concrete data we will be sending. Everything else will remain the same no matter what datatype we are sending.

Encoding the data is done through a helper class called MessageWriter. First, we instantiate the writer, passing the total size of the data into the constructor. Then we use writer's write methods to encode the data. Since we are sending two string we use write_string twice, but the MessageWriter class also supports other base types. Once wi have written all the data, we return it as a shared pointer to the message.:

```
template<> inline shared_ptr<pair<string, string> >
echolib::Message::unpack<pair<string, string> >(SharedMessage message)
{
    MessageReader reader(message);

    string user = reader.read_string();
    string text = reader.read_string();

    return make_shared<pair<string, string> >(user, text);
}
```

The second function: unpack, takes care of the reverse operation - transforming the received binary data into a c++ object. It's similar to the pack function, onlky it uses the helper class MessageReader to read the binary data.

If we want to use the library, we first need to establish a connection with the central daemon process. This is done by creating a SharedClient object to which we pass the socket path that we specified when running the daemon. In this case, the socket path is provided to the program as an argument:

```
SharedClient client = make_shared<echolib::Client>(string(argv[1]));
```

Here, SharedClient is a C++11 shared pointer, so it needs to be created with make_shared() Next, we create a publisher and a subscriber. The publisher will send the messages that the user enters, while the subscriber will display received messages on screen:

```
TypedPublisher<pair<string, string> > pub(client, "chat");
```

When creating the publisher, we need to specify the client we defined earlier, a channel name (in our case chat) and the type of a message, which is provided as a template argument. Our messages will consist of two strings (one for the

name of the user and one for the user's message), so the type is pair<string, string>. We need to define the pack and unapck methods for every class we want to use, as we did above:

```
TypedSubscriber<pair<string, string> > sub(client, "chat", chat_callback);
```

Creating a subscriber is almost the same, except we also need to provide a callback function. In this case, the function in called chat_callback and is defined on top of the source code:

```
function<void(shared_ptr<pair<string, string> >)> chat_callback =
[&](shared_ptr<pair<string, string> > m){
        const string name = m->first;
        const string message = m->second;
        std::cout<< name << ": " << message << std::endl;
    };
```

This is a C++11 lambda function that takes one argument (which is a pointer ro the same as the type of the message, ie. shared_ptr<pair<string, string>>) and returns void. The message we receive will be of type shared_ptr<pair<string, string>>, so we can call methods directly on the received message m to extract the first and second component and print them on the screen. This function will be called everytime we receive a message. Now we know how to handle received messages, but we also need to actually send our messages:

```
std::thread write([&](){
        while (client->is_connected()){
            string message;
            std::getline(std::cin, message);
            pub.send(pair<string, string>(name, message));
        }
    });
```

We will handle message sending on it's own thread. Sending a message is as simple as calling the send() method of the defined publisher, which will take one argument of the type we defined when creating the publisher (so pair<string, string>). As such, the only thing we need to do to send the message is:

```
pub.send(pair<string, string>(name, message));
```

The final thing left to do is define when we want to process the received messages. Since this is a simple chat program, we will simply continuously process them in the main thread:

```
while(client->wait(10)){
        std::this_thread::sleep_for(std::chrono::milliseconds(10));
    }
```

Every time we call client->wait(), the communicator will collect all messages send to the subscribers that were defined with it and call their callback function. Since we don't wand to completelly overload the communicator, we add a small delay into the loop. We don't have to worry about manually disconnecting the publisher and subscriber from the daemon, since this will be handled automatically by their destructors.

CHAPTER 5

Getting started - Advanced concepts

5.1 Watchers

Sometimes, you want your programs to listen to changes on a particular communication channel, for example when somebody subscribes or unsubscribes for a channel. This can be useful when you want to, for example, only start sending data once you know somebody is listening to the channel. This can be accomplished through the concept of wacthers. Watchers work similarly to ordinary subscribers, except that they do not actually read messages sent to a channel, but simply watch for changes on the channel. Interaction with changes on the channel is still accomplished with callback methods, similarly to the way they are used in ordinary subscribers.

Currently, this feature is still a work in progress, and only one type of a watcher is implemented directly: the SubscriptionWatcher, which wacthes for new subscribers on a given channel. Everytime a new subscriber subscribes, it fires a callback with the number of current subscribers on a channel. In c++, it can be used similarly to ordinary subscribers like this:

```
function<void(int)> subscribe_callback = [&](int m){
    std::cout << "Total subscribers: " << m << std::endl;
};

SubscriptionWatcher watch(client, channel_name, subscribe_callback);
```

5.1.1 Custom Watchers

If you want to define more complex watchers, you can define a subclass of the core Watcher class and override the on_event method. For an example, let's look at how the SubscriptionWatcher is implemented:

```
class SubscriptionWatcher : public Watcher {
    public:
        SubscriptionWatcher(SharedClient client, const string &alias,
                            function<void(int)> callback);
        virtual ~SubscriptionWatcher() {};
        virtual void on_event(SharedDictionary message);

    private:
        function<void(int)> callback;
    };

SubscriptionWatcher::SubscriptionWatcher(SharedClient client, const string &alias,
                                         function<void(int)> callback):
Watcher(client, alias), callback(callback) {
}
```

```
void SubscriptionWatcher::on_event(SharedDictionary message) {
    string type = message->get<string>("type", "");
    if (type == "subscribe" || type == "unsubscribe" || type == "summary") {
        int subscribers = message->get<int>("subscribers", 0);
        callback(subscribers);
    }
}
```

The on_event function takes one argumet, a pointer to a Dictionary which contaions information about something that has happ

- Summary: misc. event. Currently fires when another watcher is added to the channel
- Subscribe: a subscriber has subscribed to the channel
- Unsubscribe: a subscriber has unsubscribed from the channel

5.2 Chunked messages

Splitting large messages into several smaller chunks can improve the performance tranferring data. To make the process of splitting the data easier you can enable Chunked messaging in the publisher and subscriber classes. To do this, simply pass true as the second template argument when creating a publisher or subscriber, like this:

```
TypedPublisher <data_type, true> pub(client, channel_name);
TypedSubscriber<data_type, true> sub(client, channel_name, callback);
```

This will automatically split sent messages into smaller chunks and merge them on the receiving end, which can improve performance, particularly on large messages.

5.3 Extending subscribers and publishers

Instead of defining types and using TypedPublisher and TypedSubscriber you can directly extend the Publisher and Subscriber or their chunked variants classes for more control. Let's examine the OpenCV example to see how we can accomplish this:

```
class ImageSubscriber : public ChunkedSubscriber {
public:
    ImageSubscriber(SharedClient client, const string &alias, function<void(Mat&)> callback) :
    ChunkedSubscriber(client, alias, string("opencv matrix")), callback(callback) {
    }

    virtual ~ImageSubscriber() {
    }

    virtual void on_message(SharedMessage message) {
        try {
            MessageReader reader(message);
            int type = reader.read_integer();
            int cols = reader.read_integer();
            int rows = reader.read_integer();

            Mat data(rows, cols, type);

            //copy data simply copies binary data from it's current position of
```

```
            //length in the second argument into the first argument
            reader.copy_data(data.data, data.cols * data.rows * data.elemSize());

            callback(data);
        } catch (echolib::ParseException &e) {
            Subscriber::on_error(e);
        }
    };

private:
    function<void(Mat&)> callback;
};
```

The Subscriber is the one that receives messages, so you will need to override the on_message method to properly transform the received data before passing to the callback. Just like in the chat example, you can use a MessageReader to parse data then pass this data to the callback. You will also need to set the type of the variable the callback will be operating on.:

```
class ImagePublisher : public ChunkedPublisher {
public:
    ImagePublisher(SharedClient client, const string &alias) : ChunkedPublisher(client, alias, string

    virtual ~ImagePublisher() {}

    bool send(Mat &mat) {
        shared_ptr<MemoryBuffer> header = make_shared<MemoryBuffer>(3 * sizeof(int));
        MessageWriter writer(header->get_buffer(), header->get_length());
        writer.write_integer(mat.type());
        writer.write_integer(mat.cols);
        writer.write_integer(mat.rows);

        vector<SharedBuffer> buffers;
        buffers.push_back(header);
        buffers.push_back(make_shared<MatBuffer>(mat));
        shared_ptr<Message> message = make_shared<MultiBufferMessage>(get_channel_id(), buffers);

        return send_message_internal(message);
    }

private:

    class MatBuffer : public Buffer {

        public:
        MatBuffer(Mat& mat) : mat(mat) {
            mat_length =  mat.cols * mat.rows * mat.elemSize();
        }
        virtual ~MatBuffer() {};

        virtual ssize_t get_length() const
        {
            return mat_length;
        }

        virtual ssize_t copy_data(ssize_t position, uchar* buffer, ssize_t length) const
        {
            length = min(length, mat_length - position);
            if (length < 1) return 0;
```

```
            memcpy(buffer, &(mat.data[position]), length);
            return length;
        }

        private:
        Mat mat;
        ssize_t mat_length;
    };
};
```

The publisher will be the class that sends your message, so it needs to implement methods to transform your object oriented data into a binary representation. This binary representation consists of one or more binary buffers represented by the Buffer class. Looking at the custom buffer implementation of MatBuffer, we can see that a Buffer class needs to implement two methods: get_length (which returns the size in bytes of the message) and copy_data(), which copies the binary representation of the data to the provided buffer argument. Since this message can be split into multiple chunks (for the above chunked messaging), it needs to be able to convert arbitrarily sized portions of our object into a binary representations. This portions are represented by the position (which indicates the start of the chunk) and length (which indicates the length of the chunk) arguments. The copy_data() function needs to copy the chunk of data into the provided buffer argument and returns the number of bytes written.

The actuall subscriber only needs to override the send() method, which takes in the object that we want to send and transforms it into a binary message. In this case, the message is split into two buffers that represent two parts of the message: the header and the body. The header uses MessageWriter to encode the data, similarly to how it was used earlier, while the data buffer uses the custom MatBuffer class described above to encode the actual data. After both buffers are written, they are simply encoded into the message. Since we are using multiple buffers, we can use the MultiBufferMessage class to easily represent out message. After the message is created, we need to send it with the send_message_internal() function.

CHAPTER 6

For developers

6.1 Project structure

6.1.1 Daemon

The daemon is the central process that runs in the background, implemented in the daemon.cpp file. The main event loop first reads all pending epoll events with the epoll_wait() call. Then, it iterates over the events and performs one of the three actions based on the type of the event:

- If it's an error event, it prints an error report and terminates the daemon
- If it's an event on the daemon's listening socket, this indicates that it's an incoming connection from a new client. In this case, the daemon accepts the connection and registers the socket with epoll so that it can receive events from it
- If it's an event on a socket belonging to a client, the daemon notifies the router class that a message has to be parsed.

6.1.2 Routing

The the classes in the routing.h file are used by the daemon to properly receive and transmit messages. The header file defines three classses. The Channel class is the internal representation of a channel with methods to add or remove subscribers and watchers and send messages to all channel subscribers. The Client class is used by the daemon to represent connected processes. The router class is used to keep track of all the clients and the channels they are connected to, as well as to handle the automatic transmission of received messages to their proper recipients.

6.1.3 Message

The message.h header defines various message types and classes used for encoding and decoding of messages.

6.1.4 Client

The client.h header defines classes that are used by echo's users to connect to the daemon. It defines the basic client, subscriber and watcher classes that are used to connect to and communicate with the daemon. It also defines the advanced ChunkedSubscriber variant of the subscriber class that allows for the automatic splitting of messages.

6.1.5 Datatypes

The datatypes.h header provides typed variants of the publisher and subscriber classes.

6.1.6 Python

Python.cpp uses pybind11 to define python bindings of the library

6.1.7 Debug

The debug.h header defines helper methods used for logging and debugging of applications that use echo.