Quickmsg Tutorial

Jason L. Owens

March 9, 2016

Contents

1 Introduction

Quickmsg is a small, simple library that wraps Zyre, another small library built on top of the ZeroMQ messaging toolkit. Zyre provides functions to implement dynamic group communication between a set of cooperating processes on a local network without a lookup service. Quickmsg adds two standard communication patterns to the base Zyre (many-to-many) functionality: one-to-many publish/subscribe (pubsub), and many-to-one synchronous client/server procedure calls.

Quickmsg is written in C++, but has a C bindings which we use with SWIG 3.0 to create wrappers for Python, Java, and C# (in progress). Quickmsg also provides an initial Common Lisp binding using CFFI with the C bindings. In all cases, we've tried to create the most natural interface for the given the language.

2 Networking

Quickmsg is designed to make creating local network applications simple. Simple to code, and simple to use. All messages are based on strings (Quickmsg does not care what you put in your strings), and as long as topics are known to each participant, no IP addresses, domains, or ports are required. Zyre utilizes broadcast beacons to support discovery and then manages the available groups and peers available on the network (including their addresses and ports).

Natively, Zyre supports a group communication model where each member is a peer and may join some set of groups and communicate to all other members of a group through a single `shout()` function. Quickmsg adds support for two additional models [1]: publish/subscribe and client/server.

[1] While the goal is for Quickmsg to support the standard group model through the use of the `GroupNode` class, it is not yet well implemented.

2.1 Publish/Subscribe

The publish/subscribe model handles communication environments that don't require synchronous procedure calls and focus more on disseminating information. As an example, consider a sensor component that publishes data like ambient air temperature at a given location, or images from a camera. There may be zero or more consumer components that want that information; but the sensor component doesn't care! In this case, the sensor *publishes* the data as a message on the appropriate topic (”temp\$_{\text{sensor}}$\042”) and other algorithms can *subscribe* to the ”temp\$_{\text{sensor}}$\042” topic and receive the data as they are produced.

2.2 Client/Server

The client/server model is familiar to most everyone that uses the internet.

3 API Overview

Table of the main classes, descriptions, and important functions for implementating a Quickmsg system...

4 Basic Usage: C++ Client/Server

4.1 Basic client implementation

We begin our code-based tutorial with the simplest client implementation: just a single function, `main()`, that uses Quickmsg client calls. In this case, we do not even need to subclass the `Client`, since we primarily use the single `calls` method.

```
 1 int
 2 main(int argc, char** argv)
 3 {
 4   quickmsg::init("test_cpp_client"); // Initialize the system
 5   quickmsg::Client client("hello"); // Instantiate the Client on
 6                                     // the service topic "hello"
 7   std::string req("Hello");
 8   for (int i = 0; i < 10; ++i) {
 9     if (!quickmsg::ok()) break; // Check if the client is ok to continue
10     std::string resp = client.calls(req); // Make the service call
11     std::cout << "Service response: " << resp << std::endl;
12     sleep(1);
13   }
14   return 0;
15 }
```

In line 4 we initialize the Quickmsg subsystem, providing the base endpoint name for this collection of classes[2]. This `init` function must always be called before any other Quickmsg functions are called. The results of doing otherwise are undefined.

Then, in line 5, we instantiate the Quickmsg client that will provide communication to the specified service (where the topic name, ”hello”, is given as a parameter to the `Client` constructor). The rest of the main function simply runs a loop that sends the request ”Hello” to the server ten times (line 10), first checking to see if the client has been cancelled due to an interrupt in line 9. This is necessary since due to the network implementation in ZeroMQ, Quickmsg must intercept signals.

[2]Portions of the Quickmsg system use this name to describe the internal peers. A lot of this is not exposed yet to the user-level API.

4.2 Basic service implementation

Now we introduce the smallest example of a Quickmsg service. Line 4 is the first Quickmsg function that should be called in system that uses Quickmsg. It provides the base name for any Quickmsg used within the same process and initializes the necessary resources. In this case, we also provide a string argument that tells Quickmsg which network interface it should use for communication[3]. In line 5 we instantiate the ExampleService class we've created, and finally we spin up the service in line 6. This example shows a synchronous spin process, i.e. `spin()` does not return until the process has been exited (similar to many other event-driven systems). Quickmsg also provides an asynchronous spin mechanism that launches a thread. In this case, the application becomes multi-threaded, and appropriate care must be taken when implementing the service call.

```
1  int
2  main(int argc, char** argv)
3  {
4    quickmsg::init("test_cpp_service", "eth0"); // Initialize the system
5    ExampleService svc("hello", 20); // overridden svc impl
6    svc.spin(); // Spin (doesn't return)
7    return 0;
8  }
```

In the second code excerpt, we show the implementation for the `ExampleService` class. The service is exceptionally simple: it simply counts the number of times it's been called (stored in the `msgs_recvd` variable) (line 3), prints a message to standard out when it receives a service call (line 10), returning a formatted string to the client (line 14).

```
1  struct ExampleService : public quickmsg::Service
2  {
3    int msgs_recvd;
4
5    ExampleService(std::string topic, int queue_size)
6      : Service(topic, queue_size), msgs_recvd(0) {}
7    virtual ~ServiceImpl() {}
8    virtual std::string service_impl(const quickmsg::Message* req)
9    {
10     std::cout << "Got: " << req->msg << std::endl;
11     msgs_recvd++;
12     std::stringstream ss;
13     ss << "World " << msgs_recvd;
14     return ss.str();
15   }
16 };
```

One important thing to note is that every Quickmsg constructor requires a topic string. In the case of a service, the topic is essentially the name of the service, and is used by the client to make service calls. Going back to line 5 in the `main()` function, we see that we call this service ”hello” and give it a `queue_size` of 20 elements.

Another important item to note is the use of inheritance and virtual functions to implement the behavior of the service. Line 8 demonstrates the method signature for the service$_{\text{impl}}$ virtual function. Every message

[3]This parameter is optional, but on systems with multiple interfaces, it may be necessary to explicitly specify the interface of interest.

that is sent to the service topic (it's ”hello” in this case) will end up as a parameter to this function, wrapped in the `quickmsg::Message` object. This function must return the response to the request, so spawning a thread to perform work is not useful here[4].

[4]This implies that the work should be done in a timely manner, or other requests may not be processed quickly enough. In the current implementation of Quickmsg, the implication is true (even for asynchronous spin). Future versions of Quickmsg may allow the configuration of a threadpool or asynchronous event system to handle more request in parallel. We are not sure how the interface may/may not change to accommodate it.