Examining Latency in a Basic Publish/Subscribe System

By: Nate Newsome, and Hakeem Jones

Introduction and Literature Review:

The growth of the Internet, particularly the imminent growth of the Internet of Things, has renewed interest in new communication model. This inspired us to implement a light UDP publish/subscribe broker for our final project. Our broker has the following functionalities: subscribe, unsubscribe, connect, disconnect, end to end delay measurement and ping (between the publisher and broker). In publish subscribe systems, a publisher traditionally publishes data to a central broker who relays that data to subscribers who subscribed to that type of content. Currently content is normally distinguished by: topic (messages are distinguished by an accompanying keyword/channel), content (messages are distinguished by actual content of themselves), and type (content is distinguished by structure of message) [1,2]. Of the three variants discussed: topic based, content based and type based the author states that while the content based style is very expressive, it requires sophisticated protocols that have high runtime overhead. The authors further state that the type based approach is static and primitive, but can be implemented very efficiently. Therefore they state one should generally prefer a static scheme whenever a primary property ranges over a limited set of possible discrete values, for example, stock quotes/requests [1]. Using type-based publish/subscribe ensures encapsulation of message events by declaring private data members and only accessing them through public methods. Because a type based implementation would be more natural in an object oriented language and we were most comfortable in C, we decided to implement a topic-based publish/subscribe system over UDP.

Problem Definition:

In any publish/subscribe system we want efficient transport of messages between clients. In our basic implementation we expect to see an increased latency with increases in the number of subscribers on the system. Furthermore, we predict that this latency will be affected by client proximity to the broker.

Implementation:

To implement the publish/subscribe system we extended the UDPecho code from earlier this semester. In summary, central server acts as a content broker for the system. Clients pick a mode when sending data to the server (0-publisher, 1-subscriber, 2-un subscribe). The broker records subscriptions and waits for any connected publisher to publish content to a channel. Upon receiving new content (all content is labeled by topic channel) the broker sends the data to the clients subscribed to that channel. The broker is able to disseminate each message through use of our updated message header, which has an integer for mode and topic and a time stamp. We used the ping capabilities of the UDPecho program to give a client in publisher mode the time a published message reached the broker and back. The time stamp in the message header can be used by a client in subscriber mode to measure the delay over the system from publisher to subscriber.

Client(Publisher/Subscriber):
The client script can be run in two different modes which require different sets of arguments. First is the publisher mode (mode = 0) which is similar to the original ‘UDPEchoClient’. It is run with the following arguments:
```
./client <Server IP> <Server Port> <Mode = 0> [<topic>] [<Iteration Delay>] [<Message Size>] [<Iterations>]
```

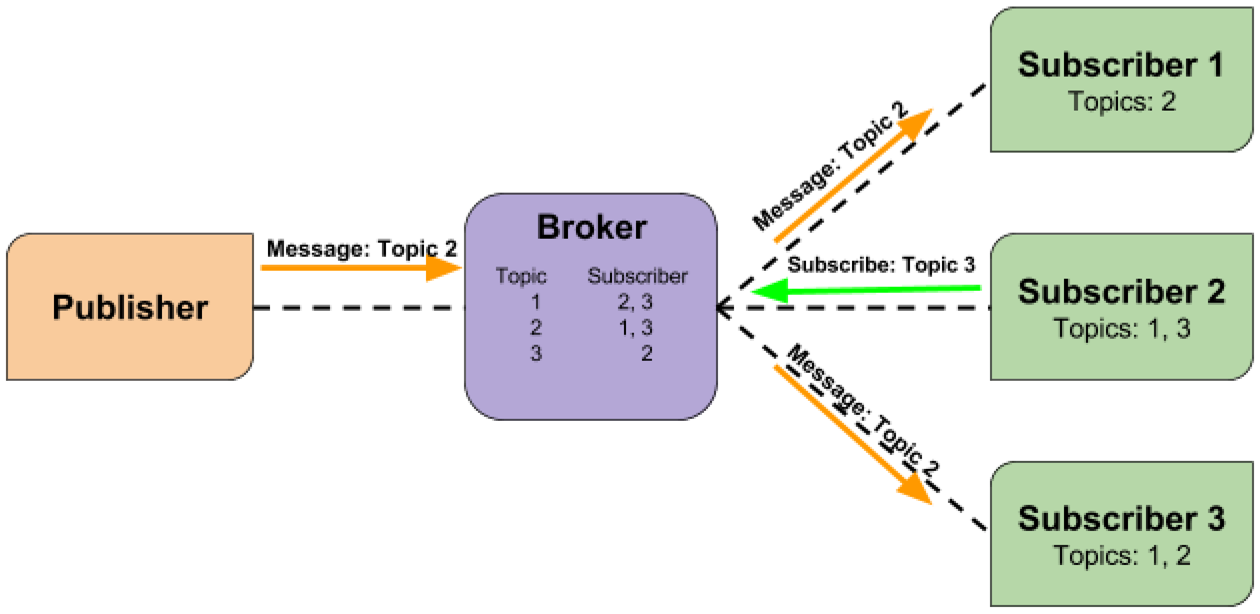
Both modes require the same first three arguments (Server IP, Server Port, Mode{0 or 1}), however the publisher requires some initial values to set up client. In publisher mode the client will send a sequence of messages for the specified topic and size. The amount of messages sent is the iterations, and the iteration delay (in seconds) is the delay between message iterations. After each message the server returns a reply and the client will calculate ping times. Next, the subscriber mode (mode =1) is run with the following arguments:
```
./client <Server IP> <Server Port> <Mode = 1>
```

Once the subscriber client is running the user is asked to input which topics it should subscribe to. After setting up the topics the client waits to receive messages from the server. To change subscriptions or unsubscribe the user can input ‘ctrl+\’ which will drop into a special mode which asks if they want to subscribe or unsubscribe to a topic. This can be done at anytime once a subscriber is running. Upon receiving messages the client calculates the trip time.

Alternatively, we have included a special subscriber mode for running simulations. This mode mode is initiated by putting the subscribed topics as arguments after the mode argument. This allows the system to run in a configuration which allows us to run multiple instances of the subscribers (via script) and have them output only the data we want to collect.

Server(Broker):
The server script is run very simply with: `./server <portNumber>`
Once it’s running the server waits to receive messages from either client. When a subscriber starts it sends special messages that tell the client that it wants to subscribe to certain topics. The subscribers are kept in a linked list which keeps track of their addresses and which topics they are subscribed to. When messages come in from publishers, the server reads the topic and forwards them to the proper subscribers.

Below you will find a diagram that depicts this system.



Methodology:
We examine the average latency of messages from publishers to subscribers to determine how it is affected by increasing numbers of subscribers. In order to accomplish this, we have set up a simulation mode for our system that runs multiple instances of the subscriber and publisher clients. The broker is run on a separate machine, after which we run a script that runs a variable number of subscribers which subscribe to one or more topics each. Once all the subscribers are connected the script will run multiple publishers which push messages to the broker for dissemination. After we've run enough messages the broker is sent a trigger to close out all of the subscribers. The subscribers then calculate the average latency of the received messages and posts that average before closing. Our simulation adds all of those times to a data file for later analysis.

Test Cases and Results:
To understand how location of the clients affects the latency we run our simulation for two different test cases. In both cases the broker is run on a department linux machine (imp4) located on campus. In the first test case, clients are run on a linux virtual machine located off campus, and in the second they are running on another department linux machine (imp5) located on campus. For each we gather the overall average latency of a variable number of subscribers (ranging from 10 to 1000).

Number of Subs.	Average Latency (Microseconds)	Dropped Subs.
10	18781.2	0
50	29093.6	0
100	30335.5	0
250	33824.2	1
500	79533.7	1
750	101864.0	17
1000	152711.1	199

Figure 1.

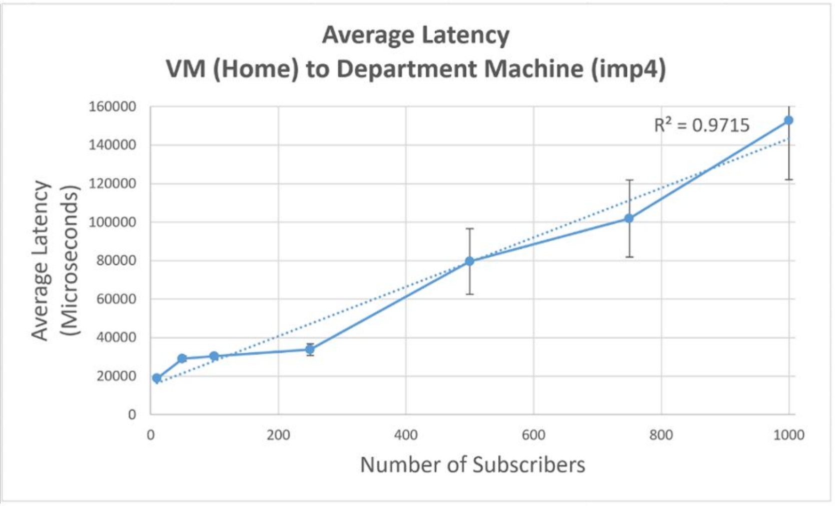


Figure 2.

Number of Subs	Average Latency (Microseconds)	Dropped Subs.
10	765	0
50	832.66	0
100	8978.44	0
250	34848.4	0
500	64265.8	0
750	100843	0
1000	128913	0

Figure3.

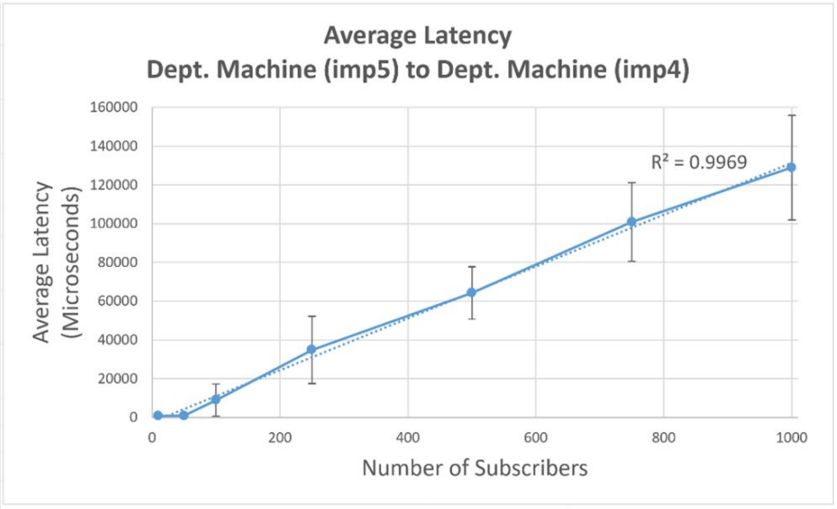


Figure 4.

We saw a linear increase in latency for each of our test cases when increasing the number of subscribers. This held true for both the home to campus configuration and the campus to campus configuration, shown in figures 2 and 4. Although, the average latencies for the home to campus configuration are more varied as we would expect from a more complicated connection. The data fits a linear regression line with an R^2 value of above 97% for both, which means that we could predict the latency for each configuration with pretty high accuracy. This data seems to confirm our hypothesis that latency will be affected by the amount of subscribers.

We performed 7 tests each, with the number of subscribers increasing for each test. The average latency for for each test is shown in Figures 1 and 3. In addition to the latency measures, we also found that with higher amounts of subscribers some subscribers did not close or report data. We are calling these “dropped subs” and we will discuss them in the next section. Latencies on the home to campus implementation were on average about 15,000 μs longer than on the campus to campus implementation. When comparing the configuration locations we saw a much higher drop rate (Dropped Subs.) on the home to campus set up than the campus to campus (figure 1, figure 3). Both of these results seem to confirm our hypothesis that client proximity to the broker impacts the average system latency.

Conclusion and Discussion:

With the imminent growth of the Internet of Things in applications such as home automation, automated vehicles, and more, there is a growing need for robust efficient systems. We have created a very basic version of a publish/subscribe system, however we can learn a lot from this simple testbed. In our tests we found a linear relationship between amount of subscribers and average latency. This begs the question, what can we change to make the system more efficient and robust?

One method for improving the system would be to use a different kind of pub/sub system such as a type-based system. While we initially set out to do exactly that, we found that it would be complicated to do that in C. It would be easier to implement a type-based system in C++ or another OO programming language.

Another interesting finding was the "dropped subs." This was not something that we initially expected. While running extensive tests and debugging our code, we were able to get the number of dropped subs down however they are still occurring with varying frequency. Initially the amount of dropped subs was much higher and we started to see drops at only 50 subscribers. Through our testing we found that the basic features of the program, such as printing status updates on messages sent and received, were causing a delay. This led to the socket being blocked for longer and messages being dropped. One possible solution is to reconfigure the broker script. As it stands, the script holds one linked list of all of the subscribers and the topics they are subscribed to. We believe that in a future iteration we could rewrite the system to hold separate linked lists for each topic, and that may have an effect on the efficiency.

Our system only shows the basics of a pub/sub system with limited topics, however with a little more work we could have a nice testbed for examining other efficiency factors. In the future we could look at setting up a multiple broker system to explore how that could impact efficiency of publish/subscribe systems.

Papers cited:

1. "The many faces of publish/subscribe" http://dl.acm.org/citation.cfm?id=857078
2. "Type-based publish/subscribe: Concepts and experiences" http://dl.acm.org/citation.cfm?id=1180481
3. MQTT-S A publish/subscribe protocol for Wireless Sensor Networks http://ieeexplore.ieee.org/xpls/icp.jsp?arnumber=4554519