Building communication modules for the
Publish/Subscribe communication paradigm using
Colored Petri nets

Hareesh Nagarajan*

CS542: Spring 2005

Dr. Sol Shatz

University of Illinois at Chicago

Contents

1 Introduction 3

1.1 A formal description of Pub/Sub 4

2 Things to consider while modeling Pub/Sub 5

2.1 Assumptions 7

3 Reasons for using Colored Petri nets 7

4 The first attempt! 8

4.1 Subscribe Module 8

4.2 The incorrect assumption set right! 11

5 The second attempt! 12

6 The third attempt! 16

6.1 The rationale 16

*hnagaraj@cs.uic.edu
6.2 A token ring among the publishers! ........................................ 17
6.3 The design of the publisher .................................................. 18
6.4 Summary ............................................................................. 19
6.5 The Publish Communication Module .................................... 19

7 The big picture ...................................................................... 22

8 Conclusion ............................................................................. 24

9 Future Work .......................................................................... 24
1 Introduction

In the CS542 course this semester, we have tried to focus on different techniques for modeling distributed systems. This has been accomplished by using Petri nets, Communicating Sequential Processes (CSP) and temporal logic. Petri nets were extended in a very interesting way in [1] to model communication primitives such as asynchronous broadcast/send/receive, synchronous broadcast/send/receive, remote method invocation etc.

If a designer designs the communication for a distributed software system using these modules then she will formally be able to verify the system that she is building. This is because the modules are internally wired using Petri nets and a variety of Petri-net related algorithms exist which can test a Petri-net for properties such as: deadlocks, liveness, progress etc.

From a software engineering perspective, the modules are available to the designer as black boxes in a CASE (Computer Aided Software Engineering) tool. The designer drags and then drops the modules onto a drawing board to essentially describe the communication that can ensue between the various nodes in her system. Once the designer has completed her design, she can then test it for the properties that were mentioned in the preceding paragraph. It is important to understand that this formal verification is only made possible because the design internally breaks down into one (or more) Petri net(s).

In this age, building distributed systems with these communication primitives can at best be described as one that has a lacks a higher level of abstraction. It would be unrealistic to expect the designer to design a large system in terms of, for example, synchronously sending a message to one or more one nodes or expecting to receive an acknowledgment from the nodes that received the message. I am not saying that it is wrong to design a system from this perspective, all I am saying is that most designers would benefit from operating at a higher level of abstraction.

The main reason for this, is the existence of new communication models for communicating in a distributed system, such as publish-subscribe (pub/sub) or point-to-point forms of message passing. Using these techniques, the job of the designer is simplified. Now, all that a designer has to think in terms of is that there are some nodes in the system which will send a message of a particular type onto the network and, that there are other nodes which may wish to receive messages of a particular type. She doesn’t have to care about how many nodes actually receive the message, or if a potential receiver of
a message suddenly went offline, or not!

These days, large scale distributed systems usually make use of middleware which have the pub/sub paradigm (or any other higher level communication paradigm) implemented within them to accomplish communication [2]. Because of this, it becomes doubly important for us to develop modules for these forms of communication.

After I spent time understanding the principal and elegance behind the communication modules described in [1], I felt the need to develop communication modules (or black boxes; which internally use Petri nets) for these higher level communication models. Once built, my goal is that these modules can be incorporated into CASE tools (that are powered by petri nets) which are used by designers of large scale distributed systems.

1.1 A formal description of Pub/Sub

Simply put, in a pub/sub system: publishers post messages with specific topics, rather than sending messages to specific recipients. The underlying messaging system then broadcasts the posted messages to all interested parties who have subscribed to the respective message (subscribers).

Now, a brief interlude into describing messaging systems will bring the right perspective in understanding the importance of pub/sub. A messaging system allows separate, uncoupled applications to reliably communicate asynchronously. The messaging system architecture generally replaces the client/server model with a peer-to-peer relationship between individual components, where each peer can send and receive messages to and from other peers [2].

The typical advantages of messaging systems over more conventional distributed computing models are the following:

- They encourage loose coupling between message consumers and message producers
- Higher degree of anonymity between the producer and consumer. The consumer does not care about the following:
  - Who produced the message?
  - When was the message produced?
  - Where on the network was the message produced?
- They permit more dynamic and flexible systems to be built.
They allow a higher degree of scalability

- They allow greater reliability due to the lack of a single point of failure.

There are two types of messaging system models which are commonly used:

- **Point-To-Point:** Messages from the sender are sent to an individual consumer which maintains a queue of “incoming” messages. The receiver then retrieves the messages from the queue.

- **Publish/Subscribe:** Here the messaging system supports an event driven model. Producers “publish” events, while consumers “subscribe” to events of interest, and consume the events. Producers associate messages with a specific topic, and the messaging system routes messages to consumers based on the topics the consumers registers an interest in. It must be mentioned that both publish and subscribe are asynchronous and non-blocking. No acknowledgment is sent either way.

2 Things to consider while modeling Pub/Sub

In figure 1 we see that there are a number of nodes in the distributed system. The nodes p1, p2, p3, ... are the publishers which publish messages of types ‘A’, ‘B’, ‘C’, ... and nodes s1, s2, s3, ... which subscribe to messages of types (‘A’, ‘C’), (‘B’, ‘C’) and (‘C’, ‘A’) respectively.

In figure 2 we can see that publisher p1 broadcasts a message of type ‘A’ onto the
medium and only the nodes s1 and s3 receive that message. The nodes then go on to do some local processing that is associated with receiving messages of that particular type. We can also observe that node s2 does not receive the message since it does not subscribe to messages of type ‘A’.

- From figure 1 we can see that multiple nodes can subscribe to messages of a particular type and also that a single node can subscribe to messages of different types. Essentially, the publish and subscribe modules need to function exclusive of the other. The functioning of one should not depend on the other.

- The publisher of a message has no motive or need to keep a track of the number of nodes in the system which are subscribing to its messages. This is an important issue which will have some bearing on our design.

- In any distributed, decentralized system nodes can appear and disappear, and their appearance and disappearance must not bring the system to a halt. In the same way the communication modules that we build should be designed with failure in mind. That is, if a publisher node or a subscriber node were to go offline, some internal transition within the Petri net which would earlier fire should now not, not fire. Essentially, the modules need to be developed with the implicit assumption of progress.

- Since the pub/sub communication paradigm makes use asynchronous communication, our modules must also be asynchronous.
2.1 Assumptions

- For simplicity, I will make an assumption that nodes that publish messages cannot subscribe to messages and vice-versa. This restriction can be easily removed.

- As in any broadcast medium only one outstanding message can exist in the system at any point in time. Once that message has been received by the receiver it will cease to exist physically in the broadcast medium.

- Again, for simplicity, I will assume that the medium does not garble packets and we do not have to bother about retransmissions or lost messages.

3 Reasons for using Colored Petri nets

The reasons for choosing to design pub/sub modules using colored Petri nets are as follows:

- In CP-nets we are allowed to use tokens that carry data values and we are also able to assign attributes for tokens. By doing so we can distinguish one token from the other.

This is the most compelling reason I had for choosing CP-nets over regular petri nets. As we have seen in pub/sub we would need to differentiate messages based on their type and to do this differentiation we would need to test the type\(^1\) of a message. CP-nets allow us to do this test easily.

- The other reason I had for choosing CP-nets was based on the fact that they offer much more modeling power than regular Petri nets. This is because they operate at a higher level of abstraction. According to [3] this can be attributed to it having better structuring capabilities.

- CP-nets are more readable than regular Petri nets. Now initially, this had no bearing on my decision to use CP-nets, but I figured that if a designer wanted to understand the internals of the modules, she could easily do so if the underlying net was a CP-net.

I must mention that my notation for CP-nets has been heavily influenced by [4].

\(^1\)which is an attribute
4 The first attempt!

4.1 Subscribe Module

The internal Petri-net wiring of the subscribe module can be seen in figure 3. As we have seen before in pub/sub, the publisher does not have to know the number of subscribers who are subscribing to its message. All the publisher has to do is to send the message with an appropriate type onto the shared broadcast medium.

In this model I have violated these requirements and in addition to that, the model does not work as expected because of a fundamental error I made.

In figure 4 we can see that the publisher has a bunch of tokens to send one after another. Initially the publisher sends across the token (TYPE, X, “msg”) onto the broadcast medium. ts is a variable which refers to the type of the message, and as we know every publisher publishes messages of a particular type. m is a variable which refers to the content of the message.

Let us assume that we bind the variable ts of transition BROADCAST MEDIUM to the
value TYPE_X, while we bind the variable \( m \) to the value “msg”. This binding is enabled in the initial marking. This gives us the binding \(<ts = TYPE_X, m = “msg”>\). Since TYPE_X is a macro and expands to “MSG_TYPE_X” the binding essentially translates to \(<ts = “MSG_TYPE_X”, m = “msg”>\). The transition BROADCAST MEDIUM is enabled with the given binding, which means that the transition may occur. When it does occur the token ("MSG_TYPE_X", "msg") will be removed from the input place PUBLISH("MSG_TYPE_X"). Simultaneously, the 2 tokens will be added to the output places that are attached to arcs that go out of the transition BROADCAST MEDIUM. That is, the places TEMP1 (refer figure 3) inside the SUBSCRIBE modules found in the nodes S1 and S2, will get the token ("MSG_TYPE_X", "msg").

It is at this point that I made my almost flagrant assumption. It could be attributed to the fact that I got so caught up in the model that I looked beyond what is permissible. I assumed that only one of the two subscribe modules would receive the token ("MSG_TYPE_X", "msg"). Because of this incorrect assumption, we can see that the publisher sends the same token ("MSG_TYPE_X", "msg") again to make sure that the token is placed in the place TEMP1 of the SUBSCRIBE module in node S2 (assuming

\footnote{I am not aware if macro’s can be used in CP-nets but I’ve used them the same way we used it in CSP. More on this aspect follows in the upcoming sections. Using macros does not change the semantics of the CP-net}
that when the transition BROADCAST MEDIUM fired the first time, the token was placed in place TEMP1 of the SUBSCRIBE module in node S1). As a consequence of this assumption, I had to build some “logic” into the SUBSCRIBE module which would prevent the SUBSCRIBE module (for example, the SUBSCRIBE module in S1) from receiving the same message again. In addition to this fact the publisher had to send additional messages to reset the subscribers to their initial states. By resetting we mean, that the token (“MSG_TYPE_X”) has to be placed into the place TEMP2 within the SUBSCRIBE module, because this token is consumed when the SUBSCRIBE module accepts a message. Even though, this is an incorrect design, both these issues will be discussed below as it will give us some insight into my thought process.

As we saw in the preceding section, once the transition BROADCAST MEDIUM fires then the token (“MSG_TYPE_X”, “msg”) would need to be placed in the TEMP1 place of both the SUBSCRIBE modules, but since I assumed that it will only go to one arbitrary place (TEMP1) in either of the SUBSCRIBE modules, let us play along and take this fact for granted and assume that the token is only placed in place TEMP1 of the SUBSCRIBE module belonging to node S1 (refer figure 4).

On closer inspection of the arc that connects the transition BROADCAST MEDIUM to TEMP1 (refer figure 3) we see that the token is placed in place TEMP1 only if the binding of ts matches the values of macro parameters RESET or TYPE. The values RESET and TYPE are place holders and are initialized when the module is created (this can be seen in figure 4). My sole goal with macros has been to try and make the SUBSCRIBE module generic. Since different SUBSCRIBE modules, subscribe to messages of different types, I needed a way of specifying the type of the message when the designer “drops” a SUBSCRIBE module onto the drawing board. By making use of macro expansions I would not need to change the entry guard which exists within the subscribe module.

In our example the value of ts is equal to the TYPE variable so the token (“MSG_TYPE_X”, “msg”) is placed in TEMP1. With a token in TEMP1 the either of the transitions RESET or Testing For Subscribe may fire. The transition Testing For Subscribe can fire because the binding <ts = “MSG_TYPE_X”> is enabled in the initial marking of place TEMP2. It does not matter which transition fires first. If RESET were to fire first, then the guard if ts = RESET would fail and no token would be placed in place TEMP2.

If the transition Testing For Subscribe fires first, then the token (tp = “MSG_TYPE_X”) is taken out of place TEMP2 and the token (ts = “MSG_TYPE_X”, m = “msg”) is taken out of place TEMP1 and then m is put in place RECEIVING, only if the guard ts = tp
evaluates to true. After this point the transition RECEIVED can fire and the message can be processed.

I stated earlier, that I made the incorrect assumption that on firing the transition BROADCAST MEDIUM only one of the two SUBSCRIBE modules receives the token (ts, m), when in reality the message is sent to both the SUBSCRIBE modules. When the publisher now sends the same message again, we now want to make sure that the SUBSCRIBE module in node S2 receives the message and at the same time we want to prevent the SUBSCRIBE module in node S1 from receiving the message again, because it just received it and it should not receive it again. We will see that there is nothing we can do about the first requirement.

Again, let us assume that we bind the variable ts of transition BROADCAST MEDIUM to the value TYPE_X (= “MSG_TYPE_X”), while we bind the variable m to the value “msg”. This binding is enabled in the initial marking. This gives us the binding <ts = “MSG_TYPE_X”, m = “msg”>. If the transition BROADCAST MEDIUM were to fire and the SUBSCRIBE module in node S2 were to receive the message, then the events occur in the same way as they did in the preceding paragraphs.

But what would happen if the place TEMP1 in the SUBSCRIBE module in node S1 received the token (ts = “MSG_TYPE_X”, m = “msg”)? As we discussed before, we must prevent it from receiving the same message again. Thankfully, the current design prevents the SUBSCRIBE module in node S1 from receiving the same message again because the place TEMP2 would not contain the token (tp = “MSG_TYPE_X”) as it did earlier, as a result of which, the transition Testing For Subscribe would never fire. The one way to get back the token in place TEMP2 is for the publisher to send a RESET message.

But there is a bigger problem, the SUBSCRIBE module in node S1 could keep accepting tokens of type (ts, m) and would never fire and the SUBSCRIBE module in node S2 could potentially never receive a message. On closer analysis this problem can be attributed to an incorrect assumption and not to a faulty design.

4.2 The incorrect assumption set right!

If we look at figure 5, we can see that in a regular Petri net \(^3\) if place S has a token then the transition T may fire and when it does, it would result in a token being placed in places D1, D2 and D3 (refer figure 6).

\(^3\)This applies to CP-nets as well
With this concept now registered firmly in my head, we can already see how we can greatly simplify our design for building the SUBSCRIBE module and this brings us to my next attempt.

5 The second attempt!

After the preceding discussion, I came up with the following designs for the SUBSCRIBE and PUBLISH modules respectively. In figure 7 we can see the internal wiring of the SUBSCRIBE module. As we can see, the complexity of the module has been reduced greatly. Now, all we need is a guard to test if the incoming token is of the type we expect it to be. If $ts$ is equal to the $type$ then the token ($m$) is placed into the place RECEIVING. Once this occurs the RECEIVED transition may fire and the node which owns the SUBSCRIBE module could go ahead and process the message. Here, we also note that we have done away with the horrendous RESET message.

Once the designer adds the subscribe module onto the drawing board she would have to specify the type of the message for which the module is subscribing messages to. As I had mentioned earlier, with the help of macro expansion, the value of TYPE inside the guard will match the $type$ the specified by the designer.
if ts = TYPE then (m) else empty

Declarations

color DATA = string;
var ts, m : DATA;
val TYPE = type;

Figure 7: The Subscribe module

PUBLISHING

1'(type, m1) 1'(type, m2) ...
PUBLISH(type, m1, ... ) { Macro style expansion for the PUBLISH Comm. Module }

Declarations

color DATA = string;
var ts, m1, ... : DATA;
val TYPE = type;

Figure 8: The Publish module
In figure 8 we can see the internal wiring of the PUBLISH module. Again, by using macro expansion, the binding \(<\text{ts} = \text{type}, \text{m} = \text{m1}\) enables the initial marking of the place PUBLISHING. If the designer passes as arguments “A”, “message1”, “message2”, “message3” to the PUBLISH module, then \(<\text{type} = \text{“A”}, \text{m1} = \text{“message1”}\) becomes the first binding, \(<\text{type} = \text{“A”}, \text{m1} = \text{“message2”}\) becomes the second binding and so on and so forth.

At this point we can ask ourselves if the contents of the messages that will be published are known *apriori*. This is irrelevant in our example, because if it were not the case then the CASE tool used for designing the system would need to produce the bindings when the CP-net is executed by either using old test data or by randomly generating data. If we look at a publisher from a higher level of abstraction, all we see is a node that produces data at random intervals of time and then sends this data onto the shared broadcast medium. So ideally, we would like the messages from the publisher to be generated at random intervals, but I lack sufficient understanding of Petri nets to attempt such a design. With the next attempt we will also see that this is not really necessary.

In figure 9 we can see pub/sub in action. Node p1 publishes messages of type ‘A’. Node s1 subscribes to messages of type ‘C’, s2 subscribes to messages of types ‘A’ and ‘C’ and node s3 subscribes to messages of type ‘A’.

The entry guards in the SUBSCRIBE module in nodes s1 and s2 will fail and as a result of this the place RECEIVING within both these SUBSCRIBE modules will have zero tokens. This is because these modules do not subscribe to messages of type ‘A’. On the other hand, the other SUBSCRIBE modules will, at some point in time, fire the transition RECEIVED. In this way only the SUBSCRIBE modules in nodes s2 and s3 will receive the message “Hello, ”. In the same way, they also receive the message “World!”.

In figure 10 we see how this design can fail. The figure shows two publishers\(^4\) connected to the transition BROADCAST MEDIUM. The design fails in the following scenarios:

- If one of the publishers has something to publish (has a token) and the other does not (does not have a token), then the transition BROADCAST MEDIUM will never fire. This is not what we want. If a publisher has something to publish, the design, should allow the publisher to go ahead and publish it.

- If both the publishers have something to publish then each of the arcs from the transition BROADCAST MEDIUM to the SUBSCRIBE modules would have two tokens

\(^4\)The situation can be easily extended to multiple publishers
with bindings \(<ts="\text{MSG\_TYPE\_A}", m="\text{Hello, }\)" and \(<ts="\text{MSG\_TYPE\_C}", m="\text{Petri, }\)".

Now, if the \text{SUBSCRIBE} module did not have an entry guard then the \text{RECEIVING} place in each of the modules would have two tokens each. But since we do have a guard, using of the two tokens must the guard be evaluated? What must we do with the other token?

Essentially, the guard expects a single token of the form \((ts, m)\) but here we have two!

- In the very first place should not the publishers synchronize access to the shared broadcast medium? Only one publisher must publish a message at a time.

Achieving the last point that I have made above will be the driving force behind my third and finally correct attempt at designing modules for pub/sub!
6 The third attempt!

6.1 The rationale

Continuing from where we left off with the last design, what we really need here is a way to control access to the communication medium. That is, only one publisher should be allowed to send out its message onto the medium at a time. So, in effect we need to guard the medium.

The first thing that came into my mind was the \textit{n-way Resource Guardian Module} (RGD) that was described in [1]. But, then I wondered if it would be a good idea to protect the common broadcast medium with an n-way RGD. An n-way RGD has signals (or arcs) that will allow the publisher to make a request for using the broadcast medium. If the request succeeds a grant signal is asserted and the publisher can go ahead and broadcast the message. But if the grant signal is not asserted, the publisher would have to wait. After the publisher successfully broadcasts its message onto the medium it has to explicitly release control of the medium by asserting the release signal to the n-way RGD.

Now, in my opinion, an n-way RGD is an interesting solution to “safely” guard a shared
resource. But the question I asked is: *Does the n-way RGD effectively model the semantics of a shared medium?* I think the answer to that question is an overwhelming *No!* In my opinion, shared memory buses may act this way, but not a shared broadcast medium. So, now I had to think of some other technique that regulates access to the medium and prevents the problems that we saw in the preceding section.

I thought long and hard on this problem and came up with numerous incorrect designs. I then asked myself how do nodes in a local area network (LAN) communicate with each other using a shared medium? The answer that I came up with involved protocols that essentially control access to the shared medium such as Ethernet, Wi-Fi, token ring ... And lo! With Token ring I may have found a solution to the problem at hand but my problems were far from over, since I now had to come up with a CP-net based model to control access to the shared broadcast medium.

In the section below I will explore my solution based on the token ring protocol.

### 6.2 A token ring among the publishers!

We can use the fact that if a publisher has a “token”[^5] and a message to publish, then the publisher will go ahead and put that message out onto the shared broadcast medium. Then it will hand the token to the next publisher so it can try and publish a message if it has one. Now, if the publisher does not have a token but does have something to publish, then it waits till it gets hold of the token before it broadcasts the message onto the medium. And finally, if the publisher gets hold of the token but does not have a message, then it passes on the token to the next publisher.

This solution ensures:

- **Progress:** After every step, the token moves from one publisher to the next, guaranteeing progress of the algorithm

- **Fairness:** Every publisher gets the same opportunity in sending a message across the medium.

And prevents:

- **Starvation:** No publisher waits for an indefinite time if it has a message to send.

Without any further ado let us look at the design of the publisher.

[^5]: Used in the context of the Token in a token ring MAC protocol
6.3 The design of the publisher

From figure 11, we can see how things work in this new scheme of things. After understanding this design, we will have a look at the PUBLISH communication module which we will also, obtain from this very design. It must be mentioned that the SUBSCRIBE module has not changed and continues to be the one presented in figure 7.

So we start off with one token with the initial marking \(<k = \text{true}\>\) in the place Hold1. I have represented this token by a black dot only because it helps us visualize the design of the system better. Just as always we have places PUBLISHING1, PUBLISHING2, ... which generate tokens of type \((ts, m)\) where \(ts\) is the type of the message and \(m\) is the actual message. Unlike before I have not given any initial marking for these places and because I assume that the tokens \((ts, m)\) are placed into these nodes whenever the publisher has a message to publish.

So initially if the publisher at the very top had a message to send then the place PUBLISHING1 would have a token of type \((ts, m)\) and the place Hold1 would have the token of type \((k)\)\(^6\) which in turn would mean that the transition Send1 is enabled and can fire. After Send1 fires, the token \((k)\) is put back into the place Hold1 and the token \((ts, m)\) is put into the place PUT MESSAGE which exists inside the shared broadcast

\(^6\)Or the black dot
medium. After that, the transition PUSH may fire and when it does, the subscribers either accept the token \((ts, m)\) if they subscribe to the message (the entry guard would evaluate to true) or they just discard the token (if the entry guard evaluates to false). The internals of the SUBSCRIBE module can be seen in figure 7.

Now, the place PUBLISHING1 is devoid of any tokens (assuming that the publisher has not generated another message in the mean time) and the place Hold1 has a token \((k)\), so the transition Move1 may fire. This is because, on closer inspection of the transition Move1, we see an inhibitor arc from the place PUBLISHING1 to it. Since the place PUBLISHING1 does not have a token, the transition Move1 may fire. When Move1 does fire the token \((k)\) is placed into place Hold2.

Now, if the second publisher which owns the place PUBLISHING2 has a message to send across, then the same steps as what we’ve seen above are repeated otherwise transition Move2 is enabled and may fire. When it does, the third publisher gets an opportunity to send a message. If a situation arises where the second publisher wishes to send a message, then it will have to wait till it gets the token \((k)\) in its place Hold2.

### 6.4 Summary

We can now summarize a few key points from the preceding discussion:

- Only when a publisher has token \((k)\) can it even attempt to send a message onto the shared broadcast medium.
- If a publisher has token \((k)\) and does not have a message to send across, then it passes the token to the next publisher
- If two or more publishers wish to broadcast a message at the same time, then they will broadcast messages one after another, depending on the order in which they get hold of the token \((k)\)

### 6.5 The Publish Communication Module

In figure 12 we can see the PUBLISH communication module and for the sake of completeness I have presented the self explanatory Shared Broadcast Medium communication module which can be seen in figure 13.
Declarations
color DATA = string;
var ts, m : DATA;
val k: BOOL;
var ts = type;

Figure 12: Publish Module

Declarations
color DATA = string;
var ts, m : DATA;

Figure 13: Shared Broadcast Medium Module
As I briefly mentioned in the last section, I assume that the publishers generate messages on the fly. That means, the place PUBLISHING will have a token of type (ts, m) if the publisher has a message to send out and will be empty otherwise. In the communication modules described in [1] the modules received external input which went to an input place within. In the PUBLISH module that I describe here, this does not happen. In my opinion this is a matter of choice and it does not change the semantics of the pub/sub modules that are being presented here, though, it could be easily extended to support the semantics of communication modules as described in [1].

When the designer wishes to create a publisher, she must specify the type of message the publisher wishes to publish along with a boolean-argument, which if true states that the publisher, is the first publisher to be created. Just as before, a scheme of macro expansion is employed here. The boolean-argument create is necessary because if the publisher we are creating is the first publisher in the system then we would have to initialize the place Hold with the initial marking \( <k = \text{true}> \) or in other words with the black dot. This is made possible by the guard in the incoming arc to the place Hold. For subsequent publishers, the boolean argument is false and the guard would evaluate to false.

There are two important things to keep in mind:

- This “black dot” is only needed for the first publisher in the system. If we didn’t have the black dot, our system would not work. We need one token (k) to get things moving.

- If we only have one publisher then the feedback arc, which goes from transition Move to the place Hold is not required. If the feedback arc were present, then every time the publisher had a message to send (which means the place PUBLISHING has a token) then the transition Send would be enabled and when it fires the token (k) is put into place Hold. After the occurrence of this event the transition Move would be enabled and when it fires a token (k) is put into place Hold via the feedback arc. So we are potentially left with two “black dots” or tokens of type (k) in the place Hold. This is a situation we do not want. Figure 14 depicts the correct usage of the PUBLISH module in this scenario.
7 The big picture

In this section we will look at how everything comes together when the designer is creating a distributed system using the pub/sub communication modules we have presented. Figures 14, 15 and 16 show the sequence of events.

In figure 14 we can see 3 subscriber nodes s1, s2 and s3. s1 subscribes to messages of type ‘C’, s2 subscribes to messages of type ‘A’ and ‘C’ and s3 subscribes to messages of type ‘A’. With the design that we have used it is very easy for the designer to add more subscribers. Adding new subscribers does not change the internal wiring of existing components in the system. This is one of the very reasons for building communication modules (or black boxes) in the first place.

The designer then adds a publisher p1 which publishes messages of type ‘A’. Since this is the first publisher in the system, the second parameter to the PUBLISH(...) macro is set to true. We can also see that the feedback arc is left disconnected. As we saw earlier, this does not change the semantics of the system.

In figure 15 the designer adds a publisher p2 which publishes messages of type ‘C’. While creating this publisher, the second parameter to the PUBLISH(...) macro is set to false and the feedback arc connects publisher p2 to p1.
Figure 15: Adding a second publisher to the test design

Figure 16: Adding a third publisher to the test design
In figure 16 the designer adds a third publisher \(p_3\) which publishes messages of type B. None of the subscribers at this point of time in the design, subscribe to messages of this type. Again, with the design we have chosen this should not affect the internal wiring of existing components in the system. If \(p_3\) were to broadcast a message, all the subscribers just “drop” the message (their entry guards would evaluate to false).

Thus, we can see how easy it is to build a distributed system which makes use of the pub/sub communication paradigm using the communication modules that we have built.

8 Conclusion

In this course, the idea of building communication modules using petri nets (for modeling systems) fascinated me ever since it was presented in class. As I began to investigate how a software designer would model a distributed system using these communication modules, I felt that they were inadequate for building distributed systems that use advanced forms of communication such as message passing.

So with this project I decided to build communication modules for one such popular message passing paradigm called pub/sub. I hope, I have been fairly successful at building these modules.

Since messages in this form of communication rely on the contents of the message,\(^7\) I felt that, regular petri nets would be too low level for modeling the semantics of pub/sub. I could not think of an easily extensible design for building these modules using regular Petri-nets. It is for this reason that I chose to use colored Petri nets. Now, to do this I had to understand the rules and conventions of CP-nets, so, it is likely that I could have made errors in my CP-net based models. But I do hope the message that I wished to convey with these models is not completely lost.

9 Future Work

When I tried to survey\(^8\) the literature for work that has been done related to pub/sub and Petri nets I was unable to find anything that combined these two areas. I feel that if I can build a repository of pub/sub based communication modules using plain petri-nets (and maybe investigate other kinds of petri nets) and additionally understand the typical

\(^7\)Message type
\(^8\)Running multiple search queries which included the words pub/sub and Petri nets on Google!
usage of pub/sub in real world distributed systems then I would have actual results to make a publication out of.

Additionally, testing a distributed system design that has been built using my modules through a tool like Design/CPN or CPNTools will also greatly help validate the correctness and usability of my modules.

These are issues that I wish to explore over the summer.

References


