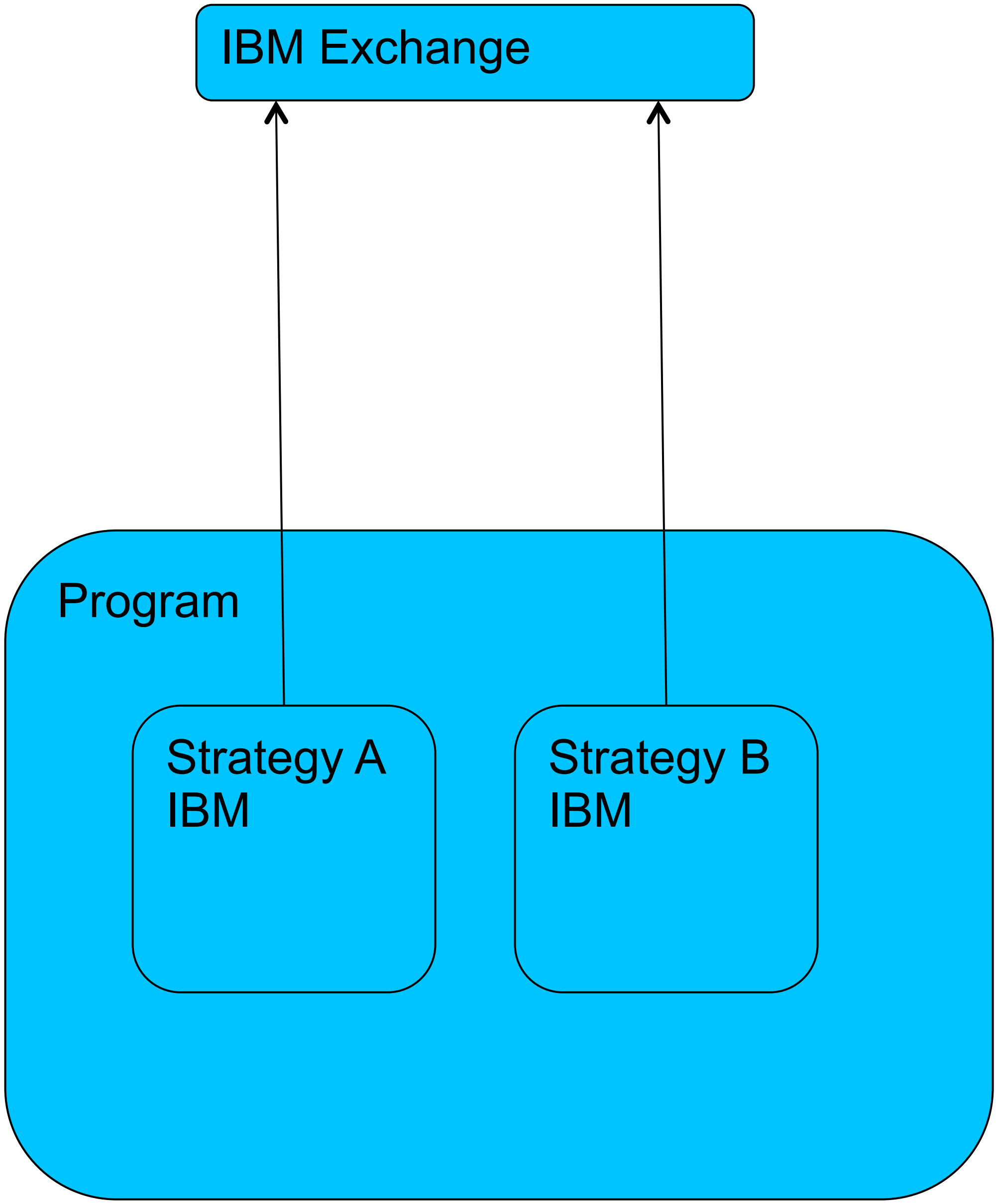
Middleware (Why, What?)

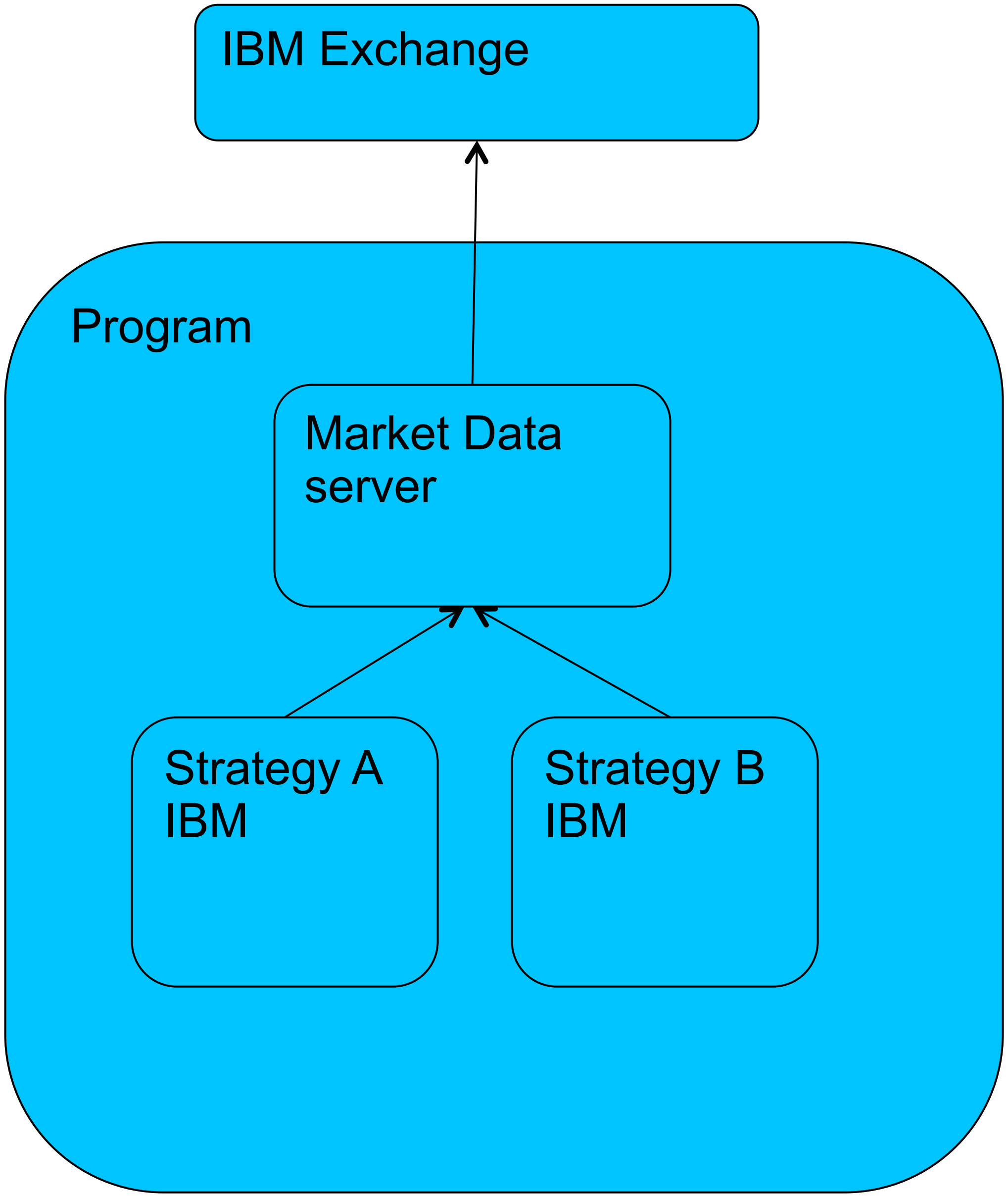
RPC, Socket Programming, JMS, ActiveMQ, AMPQ

The problem



- 2X the bandwidth
- How the strategies will get the data?

The problem



- We add a new component.
- Save on bandwidth
- Still how the strategies will communicate with the data server?

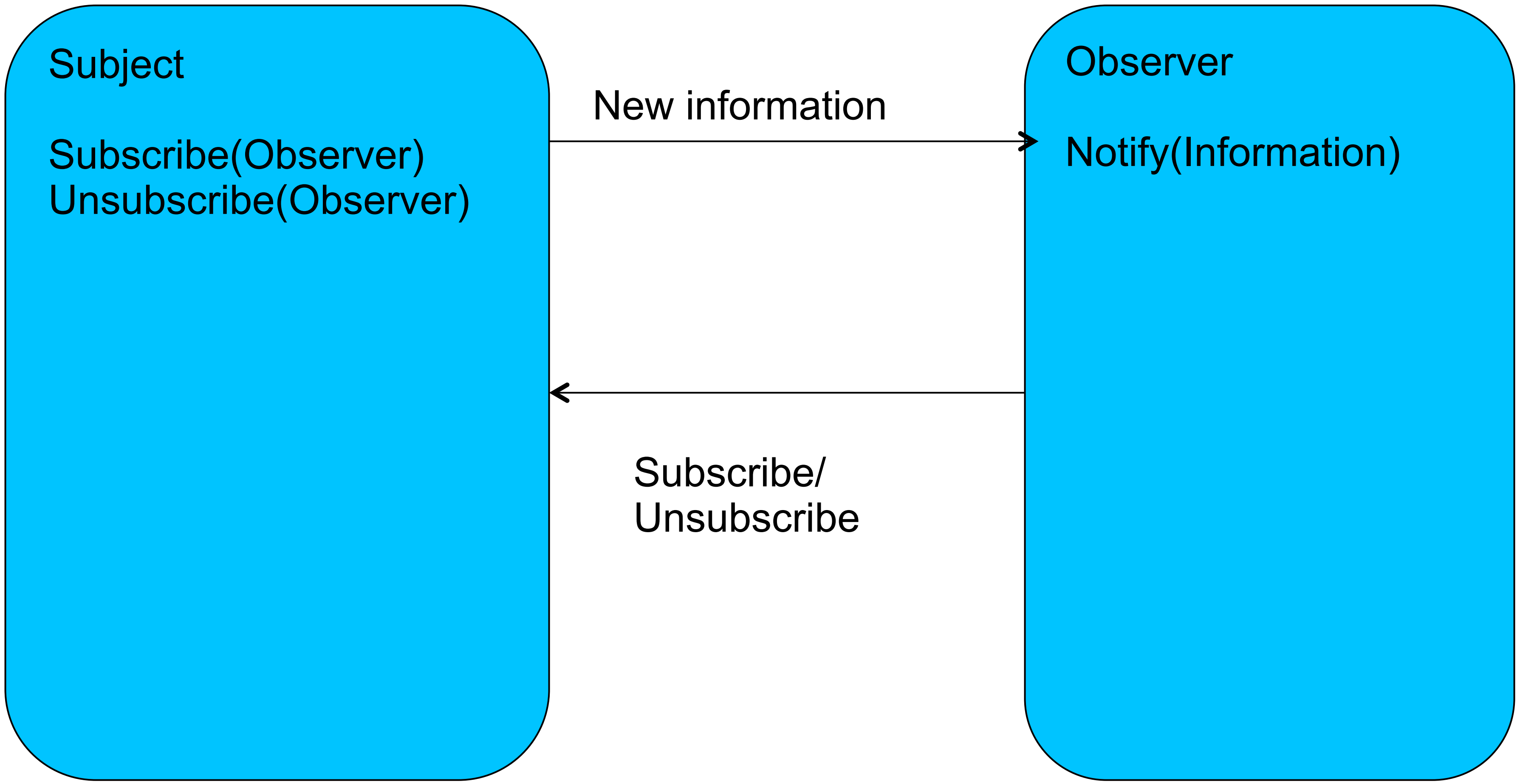
Push vs. Pull

- The strategies can use the push method
- They will continuously ask for the current data
- What is the problem with this approach.
  1. You can ask for new information every, say, 1 sec.
     - The program might miss information that occurred during the sampling period.
  2. You check for new data as fast as possible.
     - You are going to trash the server.
- Both approaches are not natural.
- Think about the general problem. A source of data generates new information in unpredictable rate.

The Observer Pattern

- The problem that we have is that we have events.
- There are multiple “independent” parts that are interested in reacting to this events. *Distributed event handling.*
- In the observer pattern we have two components:
  - Observers: The components that are interested in the events.
  - Subject: The component that generates new events.
- The subject manage a list of observers that are interested in the data.

The Observer Pattern



Subject

Subscribe(Observer)
Unsubscribe(Observer)

New information

Observer

Notify(Information)

Subscribe/
Unsubscribe

The observer pattern

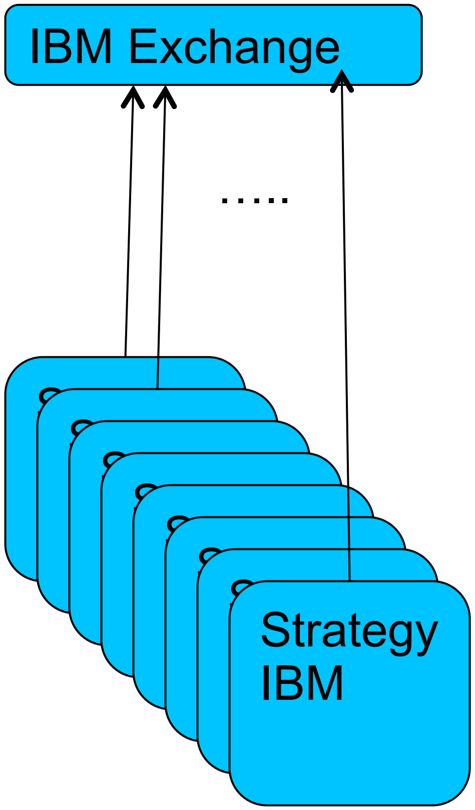
- Separation of concerns
  1. The observers are responsible for subscribing (when they are interested) and unsubscribing (when they lost interest) in the events.
  2. The subject is responsible for alerting the observers about new data when such exists.
- No trashing of the system.
- What are the pitfalls:
  - Observers can block
  - Observers can become zombies.
- Let’s see a simple example.

Market Data Server

- Currently you need about 1Gb bandwidth just for level 1 data.
- Often you run many trading applications in parallel.
- If each one consume 1GB line, you will need very very fat pipes.
- Internally (in your Data Center you have a lot of bandwidth.)
- To the outside world bandwidth is expensive.
- Often one will use an application that listen to the market data on the outside.
- This application will forward the data to all the applications.
- This application is usually called Market Data Server.
- Lets build a simple Market Data Server.

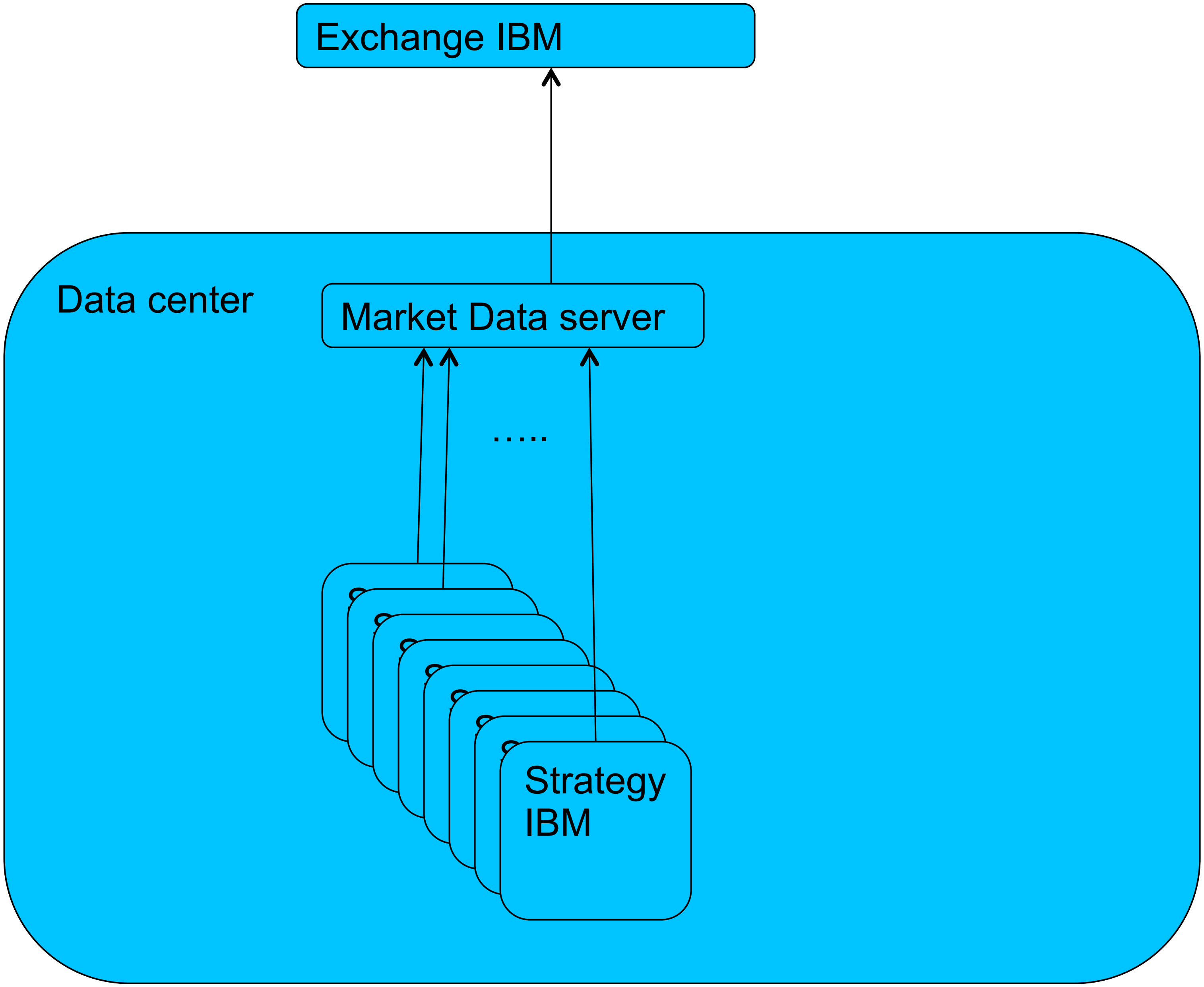
The multi-computer problem

- In reality the problem is compounded by the fact that our strategies are in different computers.
- In the language of our original problem.



- The problem is now compounded multiple times.

Market Data server



Socket Programming

- One common method for implementing Market Data server is socket programming.
- Java is very very Socket Friendly.
- What is a Socket?
- A socket is an end-point of a two way communication between two applications.
- A socket is bound to a port and can be used by the application to send data to the other side and receive data from the other side.
- Our application call for a one to many topology.
- The one is a server, and the many are clients.
- The server accepts clients and serve them data.

Socket Programming

- Java offers two basic sockets, Socket (dahh) and ServerSocket.
- These two class hide any platform dependent details of the communication layer.
- SocketServer server = new SocketServer(43432) starts a server that listen on the 43432 port.
- server.accept() will block until someone connects and return a Socket.
- What is a Socket?
- Socket provides many methods but once you have a socket two methods are important
- getInputStream(): Allow you to read from the socket.
- getOutputStream(): Allow you to write to the socket.

Socket Programming

- Everything you will write to the Outputstream will appear on the inputstream of the client (and vice versa.)
- On the client side, all you need to do:
- Socket socket = new Socket(serveraddress ,port) and you get the connection
- Let's look at a very simple application that communicate between two programs.
- What are the problems?
- Connection is fine, but we need to design a protocol.
- What happen if connection is lost?
- Who detect a problem and recover from a problem.

RPC

- The previous example shows the problem with Socket programming.
- Another popular technology is RPC (and its cousins)
- Remote Procedure Call is a very simple idea.
- Machine #1 execute a method on Machine #2.
- During the execution Machine #1 is blocked from continuing its execution.
- Once done, execution is continued on Machine #1.
- How we will apply this on our market data server problem?
- We can use RPC and the observer Pattern to deliver message.
- It is going to be Ugly (;

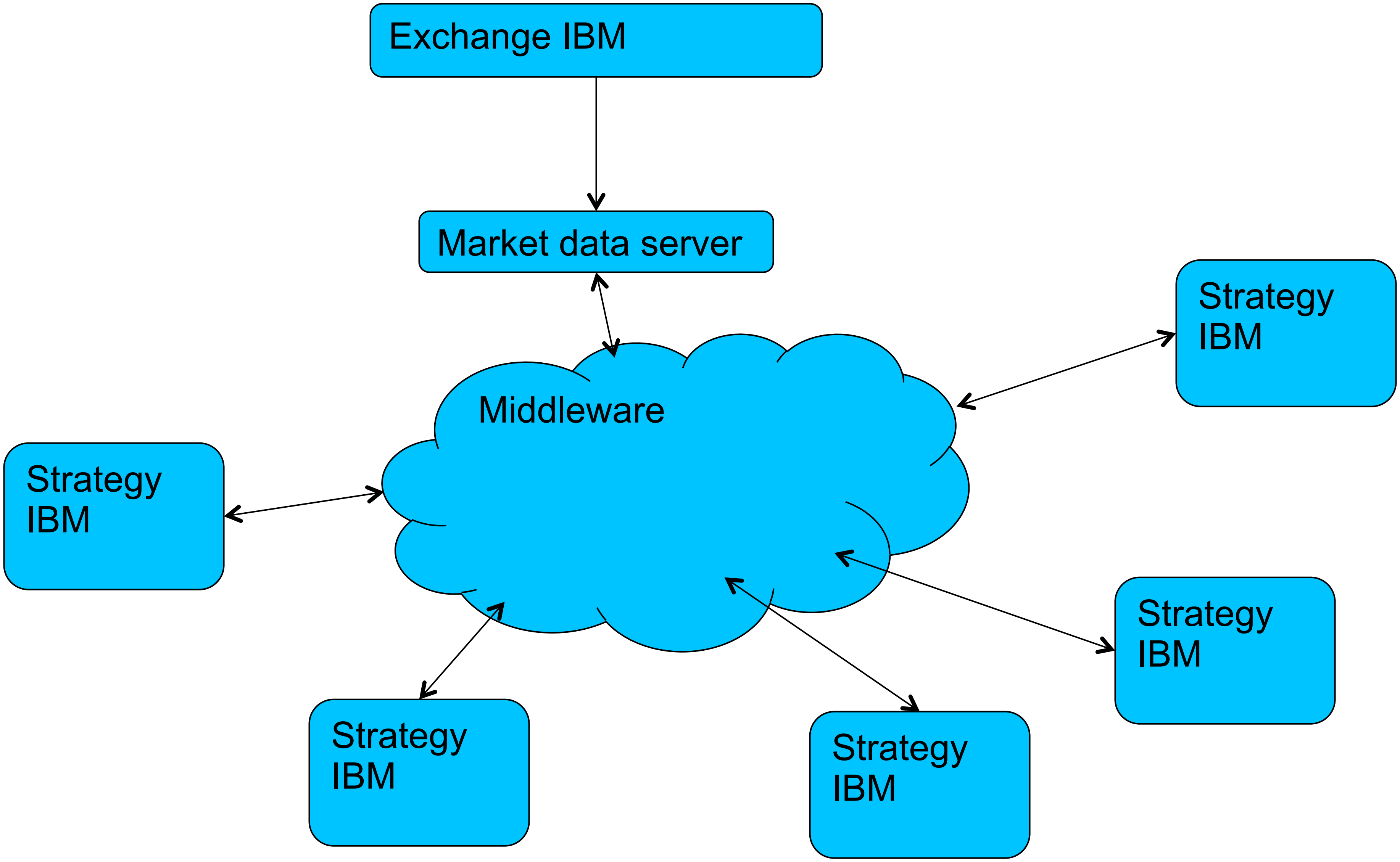
RPC

- The clients will have a method, say newMarketData(...)
- The Server will call this method on each machine.
- What are the problems?
- What is Machine #3 disappear?
- What if on Machine #12 you have a bug it takes 20sec to return?
- How we manage subscriptions?
- More than anything, the risk of blocking is large and serve as a huge incentive not to use this

Middleware

- The optimal solution is what we call Message Oriented Middleware (MOM).
- MOM is very popular idea.
- There are many implementations. TIBCO, 29West, and ActiveMQ are popular choices.
- So what MOM is?
- MOM is best described as a category of software for communication in an loosely-coupled, reliable, scalable and secure manner amongst distributed applications or systems.
- The idea is very simple. We insert a broker into our network.
- The broker receive all the messages and in an asynchronous way sends the message to its intended destination.

Middleware



Exchange IBM

Market data server

Middleware

Strategy IBM

Strategy IBM

Strategy IBM

Strategy IBM

Strategy IBM

Middleware

- The broker isolate all the interested parties from each other.
- The senders send a message and continue execution. The broker will take care of the rest.
- So why MOM is so good.
- It create a very loose coupling between applications.
- It can be used to ensure reliability.
- It can be used to persist messages.
- It can support complex routing (PoP, Multicast, and more.)
- Cons: (almost) NONE.

Middleware

- Your task is to design a MOM. Now what?
- We need to design a message. A reasonable start is to have two parts. Header and Payload.
- Header will contain the destination, etc.
- But how would you know what are the destinations?
- MOM solves this problem in a very elegant way.
- Instead of an address we can define a topic.
- This message topic is a quote for IBM or maybe a complicated problem to solve.
- The broker will deliver the message to any listener that is interested in this topic.
- We can define something like: deliver to every one that is interested or deliver only once.

Middleware

- The clients will register with the broker for topics.
- The broker will deliver all the messages that the client is interested in.
- In essence, the broker is the subject and the clients are the listeners in the Observer pattern.
- The design we just described is in the hurt of the JMS implementation.

Java Messaging System

- JMS is an attempt to define a non vendor centric API for MOM.
- This will allow you to write code that should work with many different MOM implementations.
- Each vendor provides its own implementation (and more) of the JMS.
- The Java Message Service is a Java API that allows applications to create, send, receive, and read messages. Designed by Sun and several partner companies, the JMS API defines a common set of interfaces and associated semantics that allow programs written in the Java programming language to communicate with other messaging implementations.
- JMS is asynchronous and reliable.

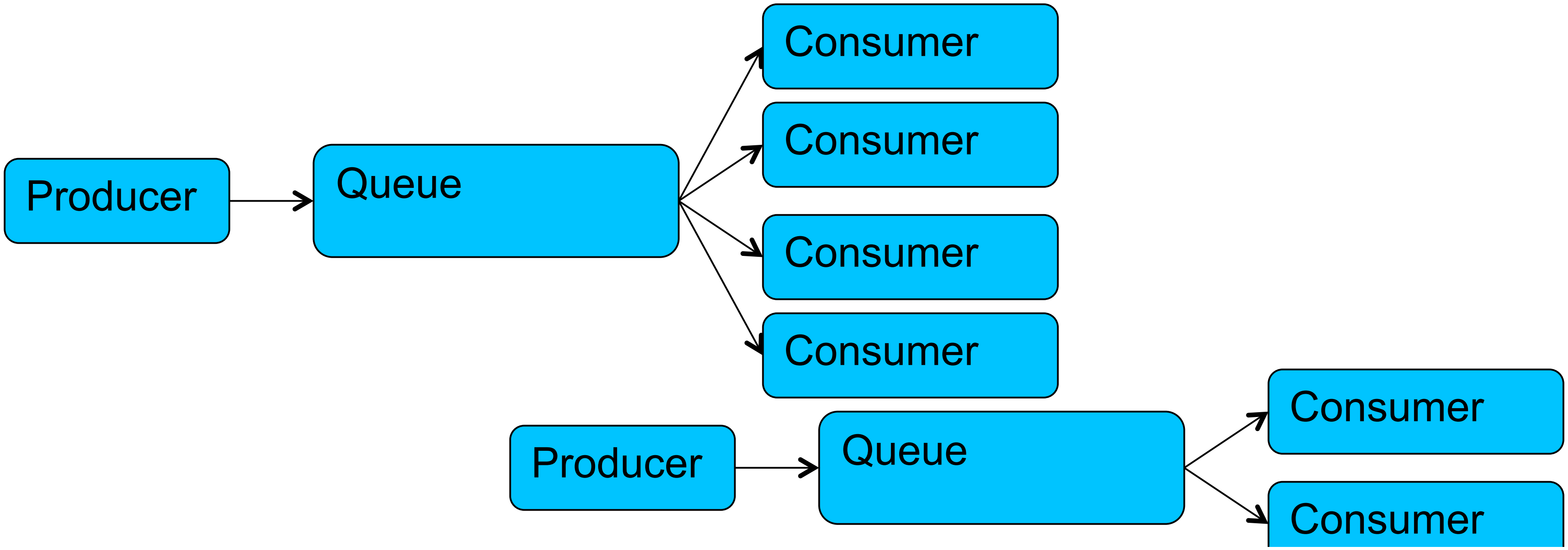
JMS - stylized

Sender

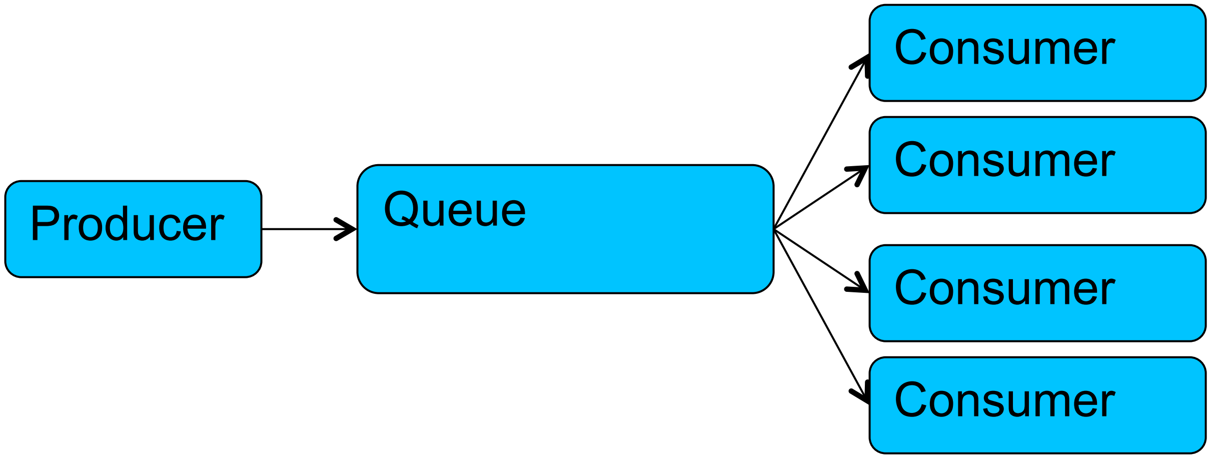
```
Session <- createSession

Topic <- session.createTopic(“xyz”)

Sender = session.createSender(topic)

Sender.sendMsg(“abc”)
```

Receiver

```
Session <- createSession()
Topic <- session.createTopic(“xyz”)
Session.listen(topic, new
        messageListener(){
                onNewMessage(msg){
                        Do something
                }
        }
)
```



JMS – Queue Model



Producer

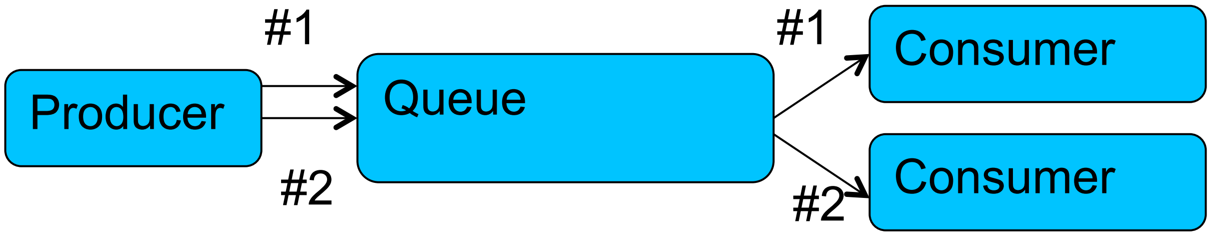
Queue

Consumer

Consumer

Consumer

Consumer



#1

#2

Producer

Queue

#1

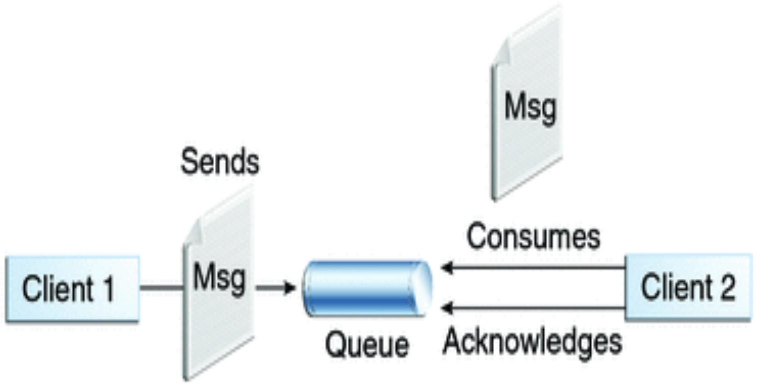
#2

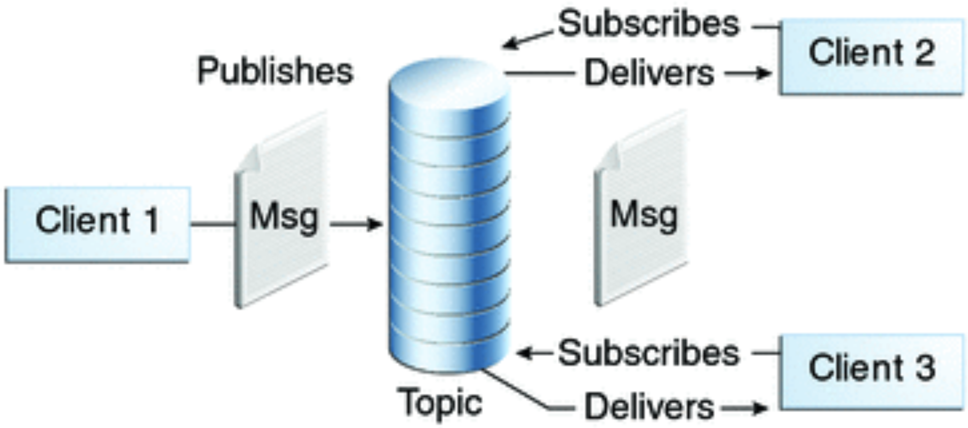
Consumer

Consumer

JMS basic Concepts

- The four basic JMS concepts are:
- JMS provider: The broker. This could be either PtP or Publish/ Subscribe.
- JMS Client: The programs that sends and receive messages.
- Messages: The actual message we send from one end to another.
- Administrative Objects: Objects that the broker use for administrative purposes.



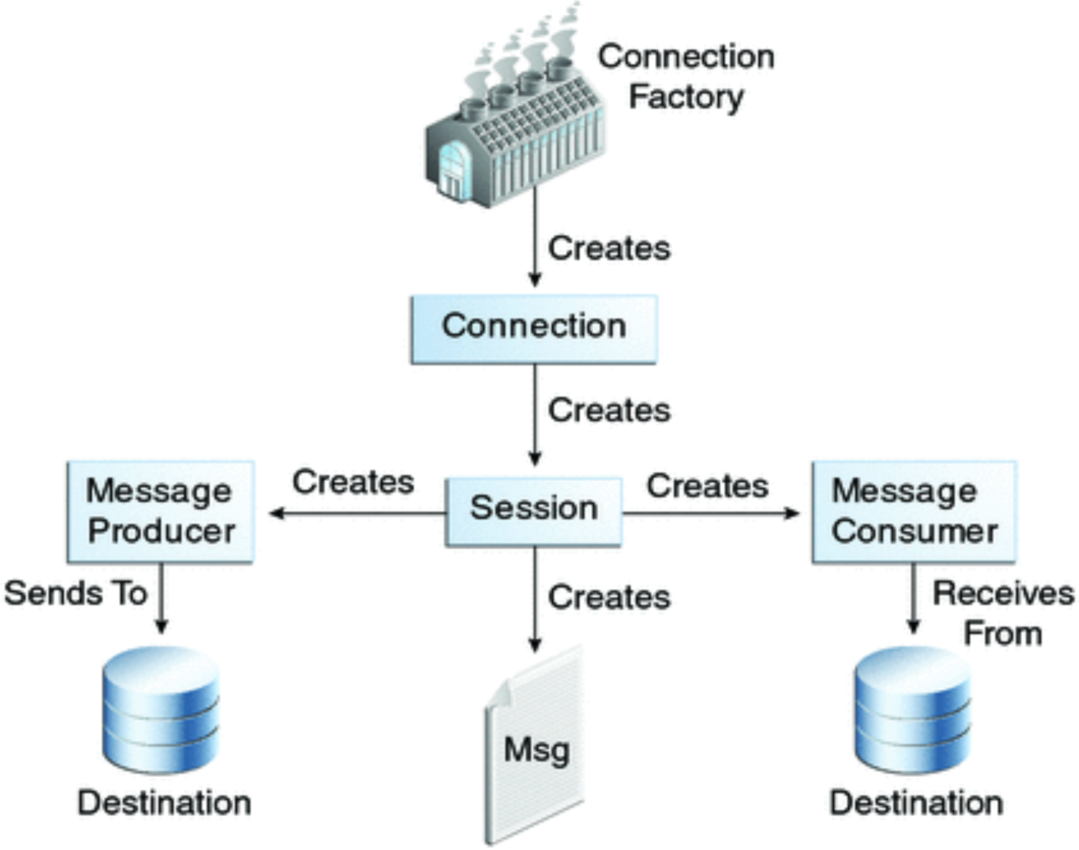


JMS Anatomy

- JMS is for sending and receiving messages.
- A JMS message has three parts. Header, Properties, and Payload.
- Headers are the basic fields that are used to “route” the message properly.
- Examples: JMSDestination, JMSExpiryMethod, JMSExpiration.
- Properties are additional application centric key/value pairs.
- Payload: Can be String, Map, Object, … (Message)
- A JMS Consumer is a the class that can be for receiving messages sent to some destination.
- They support both a synchronous and asynchronous modes.
- One can register a listener and use receive that blocks. (MessageConsumer)

JMS Anatomy

- A message producer is a class that can be used for sending messages to a destination.
- Call send and you are done. (MessageProducer)
- A session encapsulates the producer, consumer, messages, etc. (Session)
- JMS also support selectors.



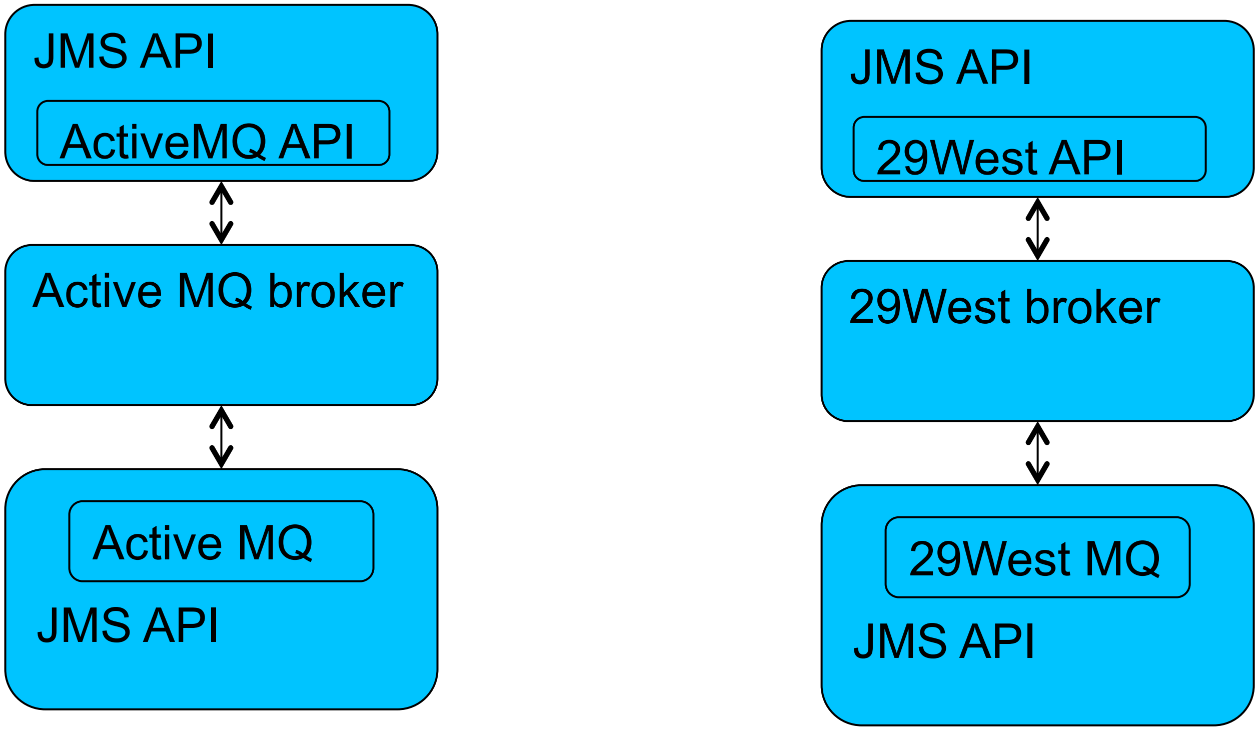
JMS – How to do it

- Acquire a JMS connection factory
- Create a JMS connection using the connection factory
- Start the JMS connection
- Create a JMS session from the connection
- Acquire a JMS destination
- Create a JMS producer, OR Create a JMS producer
- Create a JMS message and address it to a destination
- Create a JMS consumer. Optionally register a JMS message listener
- Send or receive JMS message(s)
- Close all JMS resources (i.e., connection, session, producer, consumer, etc.)

ActiveMQ

- JMS is an API. A set of interfaces.
- ActiveMQ is a great FREE MOM which is also JMS compliant.
- ActiveMQ offers many features:
- JMS Complient
- Connectivity over many protocols. (TCP, UDP, HTTP)
- Pluggable Persistence.
- Client APIs (in almost any language.)
- Broker Clustering.
- Simple Administration and tools.
- Let's build the Market Data Server.

JMS



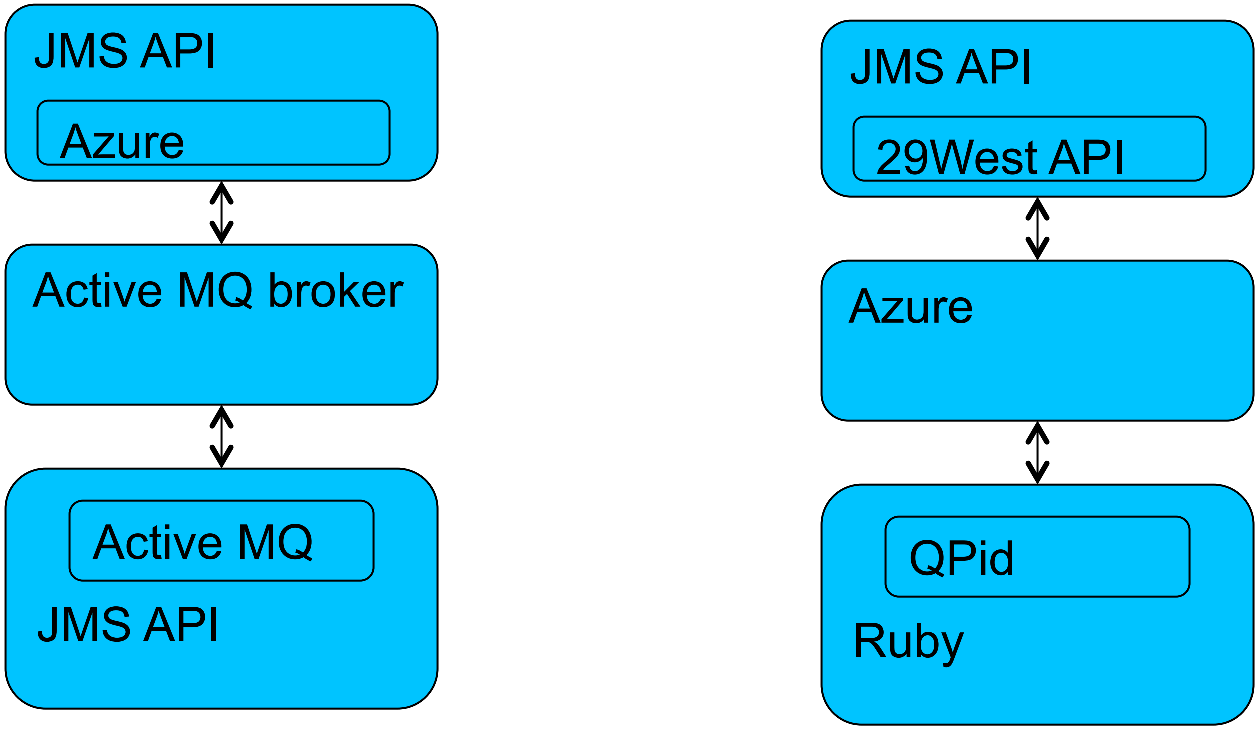
- We have to have the same broker/client along our stack.
- What happen if we want to “talk” with a ruby program?

JMS

- JMS is a JAVA API.
- No rules on exactly how to implement it.
- There is no guarantee on inter-operability.
- Some brokers allow cross platform implementation. (ActivMQ, Tibco.)
- All this drawbacks steam from one issue. JMS is an API not a wire protocol.
- If we would have defined a wire protocol we will:
  1. Allow cross platform messaging
  2. Allow different client/brokers implementation to talk with each other.
- The solution is AMPQ which was pioneered in the financial industry (lead by J.P. Morgan) in 2006.

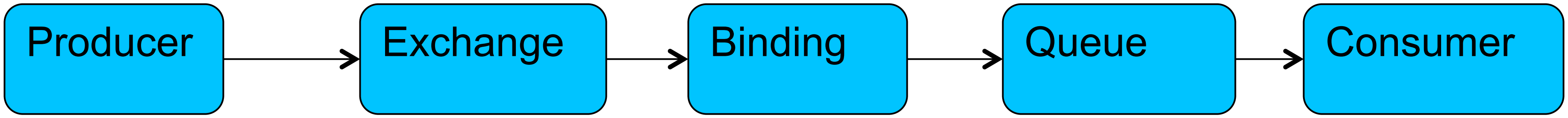
AMQP

- AMPQ is a wire protocol for messaging.
- Just like a JMS stack it has a broker in the middle.
- However, due to interoperability, an AMQP model is like.



AMQP - Model

- The AMQP model routing logic is much more comprehensive than JMS routing model.



Producer

```
Exchange = channel.declareExchane(“MarketData”)
Msg = IBM market data
RoutingKey = IBM
Channel.publish(exchange, routingkey, msg)
```

Consumer

```
Queue = channel.declareQueue(“All Market Data”)
Channel.createBinding(queue, “MarketData”,”*”)
Queue.consume(…)
```

Consumer

```
Queue = channel.declareQueue(“All Market Data”)
Channel.createBinding(queue, “MarketData”,”IBM”)
Queue.consume(…)
```