Corba Event Service
Corba Notification Service
MOM

CORBA Event Management

- CORBA event management service defines interfaces for different group communication models.
- Events are created by *suppliers (producers*) and communicated through an *event channel* to multiple *consumers*.
- Service does not define a quality of service (left to implementers).

Benefits of the OMG Event Service

- **Anonymous consumers/suppliers**
  - Publish and subscribe model
- **Group communication**
  - Supplier(s) to consumer(s)
- **Decoupled communication**
  - Asynchronous delivery
- **Abstraction for distribution**
  - Can help draw the lines of distribution in the system
- **Abstraction for concurrency**
  - Can facilitate concurrent event handling

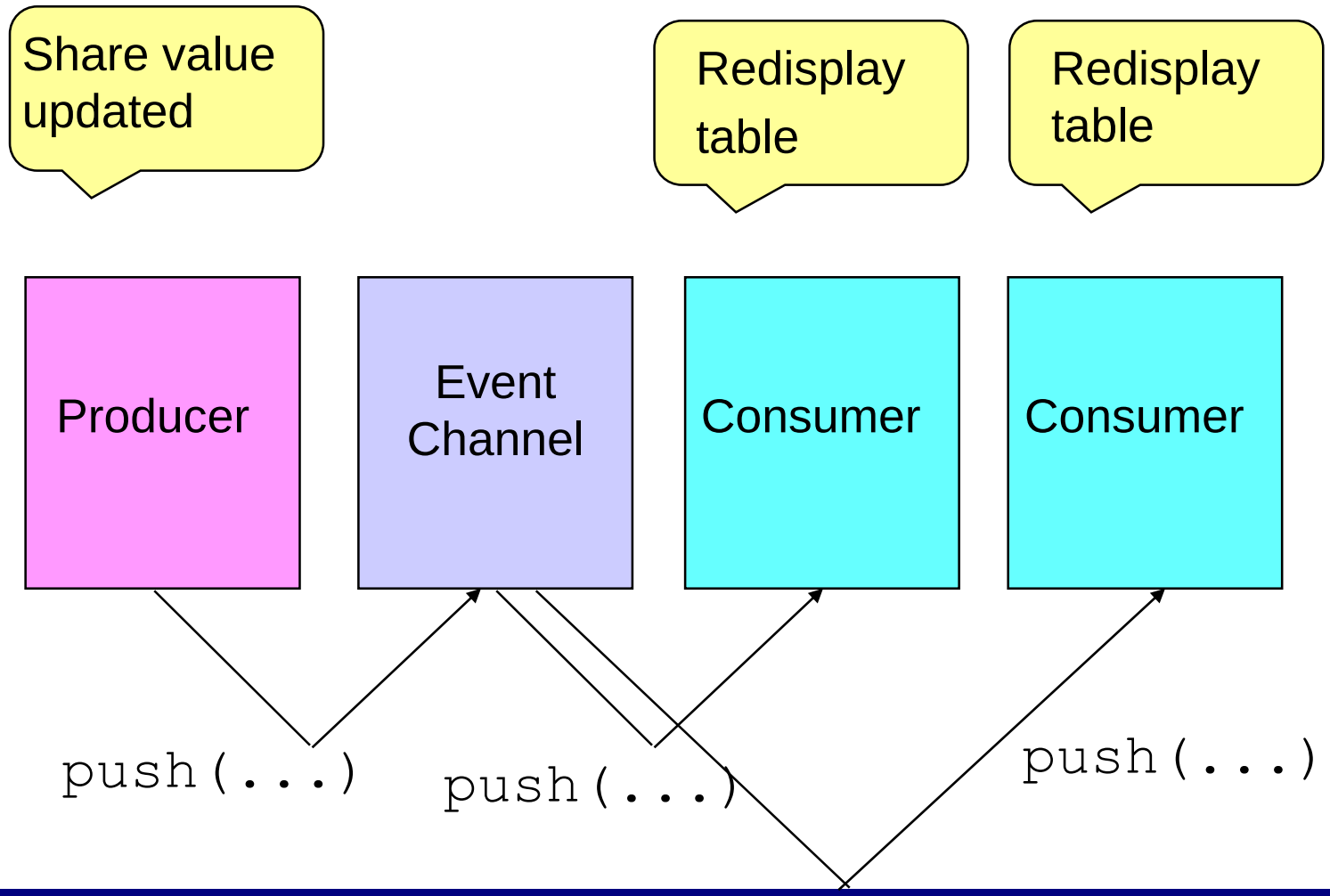
Models

- Push Model
- Pull Model
- Hybrid Model

Push Model

- Consumers register with those event channel through which events they are interested in are communicated.
- Event producers create a new event by invoking a push operation from an event channel.
- Event channel notifies all registered consumers by invoking their push operations.

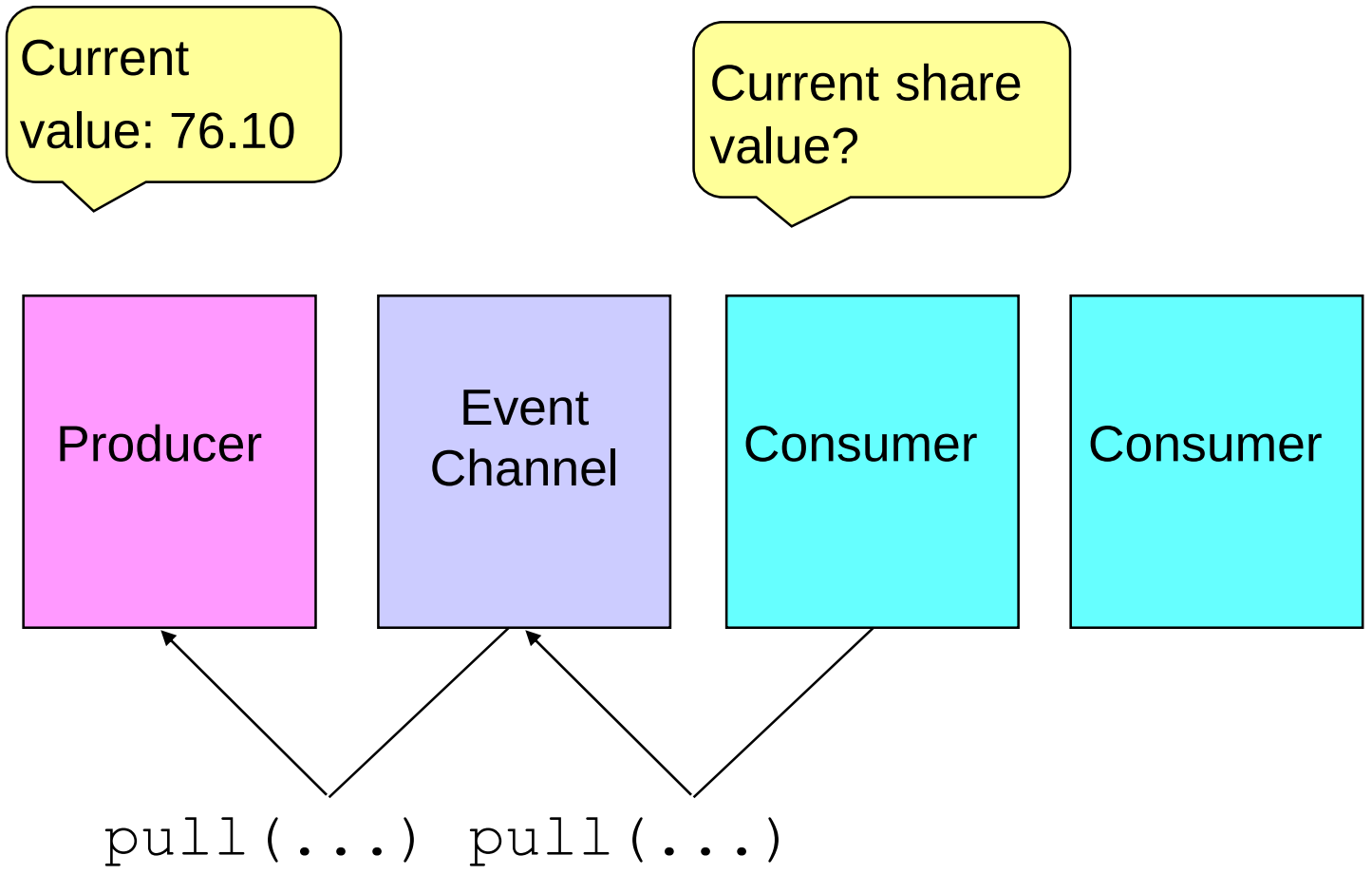
Push Model (Example)



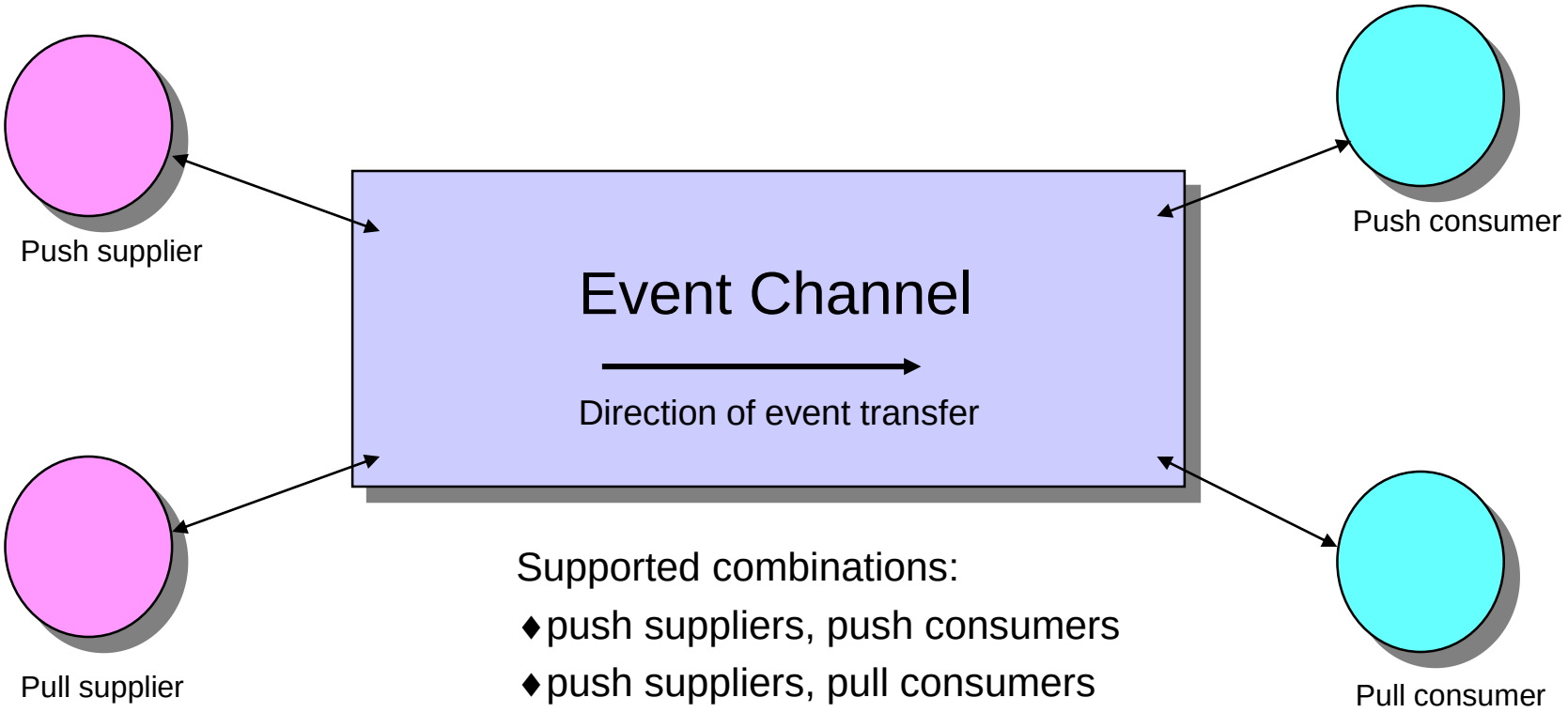
The Pull Model

- Event producer registers its capability of producing events with event channel.
- Consumer obtains event by invoking pull operation from event channel.
- Event channel asks producer to produce event and delivers it to the consumer.

Pull Model (Example)



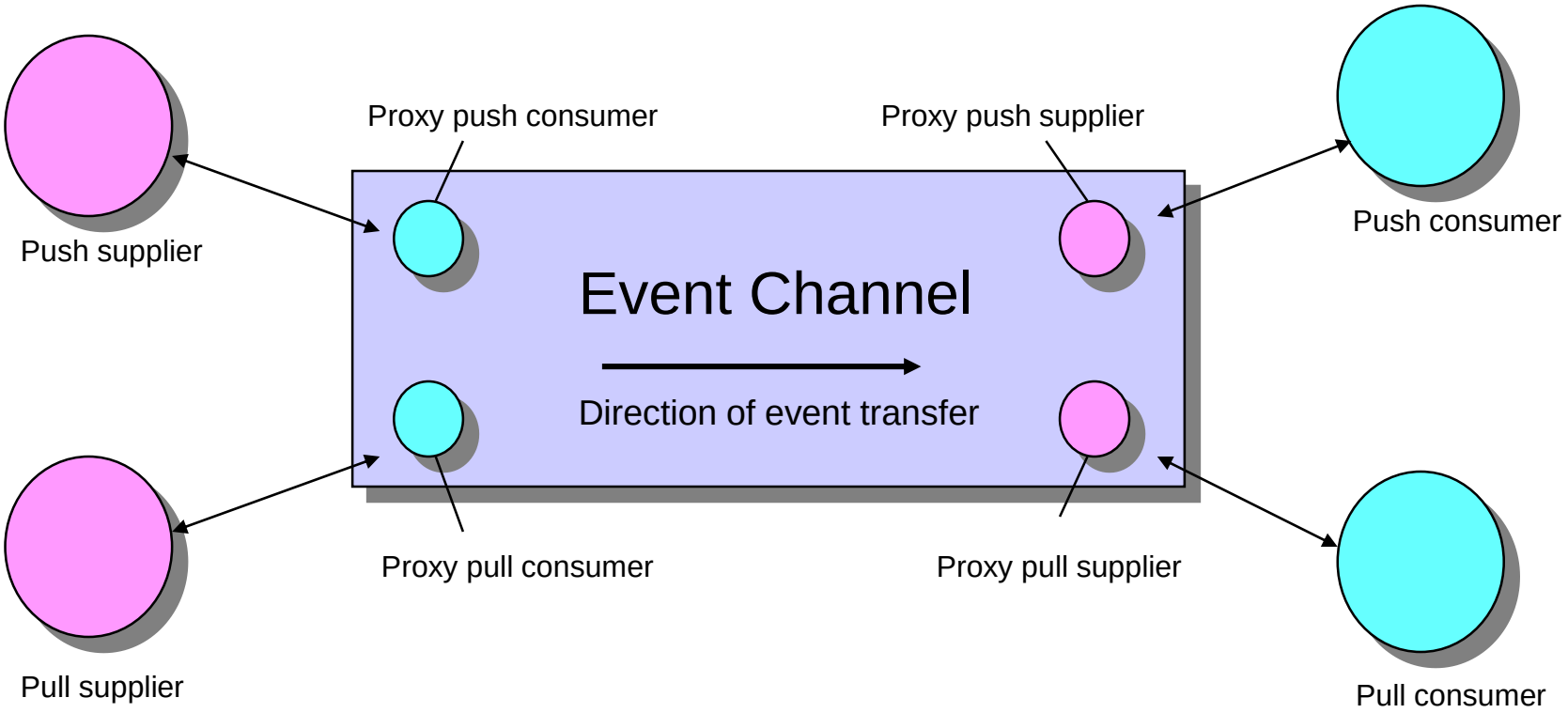
Event Channel



Supported combinations:

- push suppliers, push consumers
- push suppliers, pull consumers
- pull suppliers, push consumers
- pull suppliers, pull consumers

Event Channel (with proxies)



The Connection Process

Regardless of the model, the following steps must be taken to connect to an event channel:

1. The client must bind to the Event Channel, which must already have been created by someone, perhaps the client
2. The client must get and Admin object from the Event Channel
3. The client must obtain a proxy object from the Admin object (a Consumer Proxy for a Supplier client and a Supplier Proxy for a Consumer client)
4. Add the Supplier or Consumer client to the event channel via a connect() call
5. Transfer data between the client and the Event Channel via a push(), pull(), or try_pull() call

Reducing Client –Server Coupling with `any`

- `any` is an idl modifier that means any valid type is permissible.
- Huh?
- We are used to dealing with strongly typed languages in which the compiler does a lot of type checking.
- Weakly typed languages place the type checking burden on the programmer.

Push Supplier/Consumer

```
module CosEventComm {

    exception Disconnected{};
    interface PushConsumer {
      void push (in any data)
  raises(Disconnected);
      void disconnect_push_consumer();
    };

    interface PushSupplier {
      void disconnect_push_supplier();
    };
```

Pull Supplier/Consumer

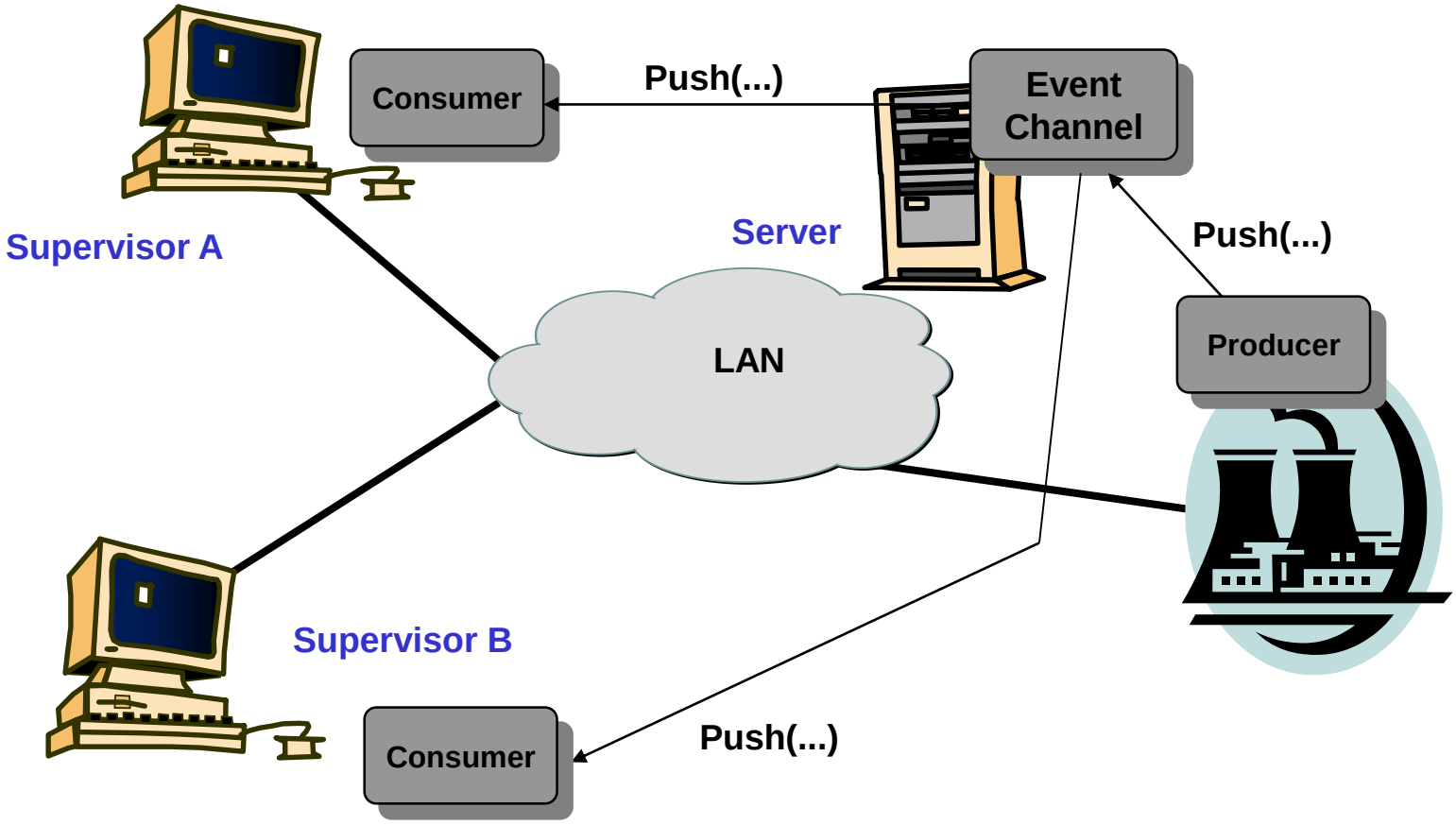
```
interface PullSupplier {
      any pull () raises(Disconnected);
      any try_pull (out boolean
  has_event)

  raises(Disconnected);
      void disconnect_pull_supplier();
    };
    interface PullConsumer {
      void disconnect_pull_consumer();
    };
};
```

How is any implemented?

- Any is a CORBA class that contains two fields, a type field and a data field.
- Any has two getter methods
  - `extract_TypeCode()`
  - `extract_Object()`
- Once the object is extracted, you use the type code to narrow the object to the type you want

Event Channel Use case



Producer pushes Any types!

```
String s = "PushSupplier says: " + ....

// Insert string into any
//
Any any = orb_.create_any();
any.insert_string(s);

// push to event channel
try{
        consumer_.push(any);
}
catch(org.omg.CosEventComm.Disconnected ex)
{
....
```

Consumer receives Any types

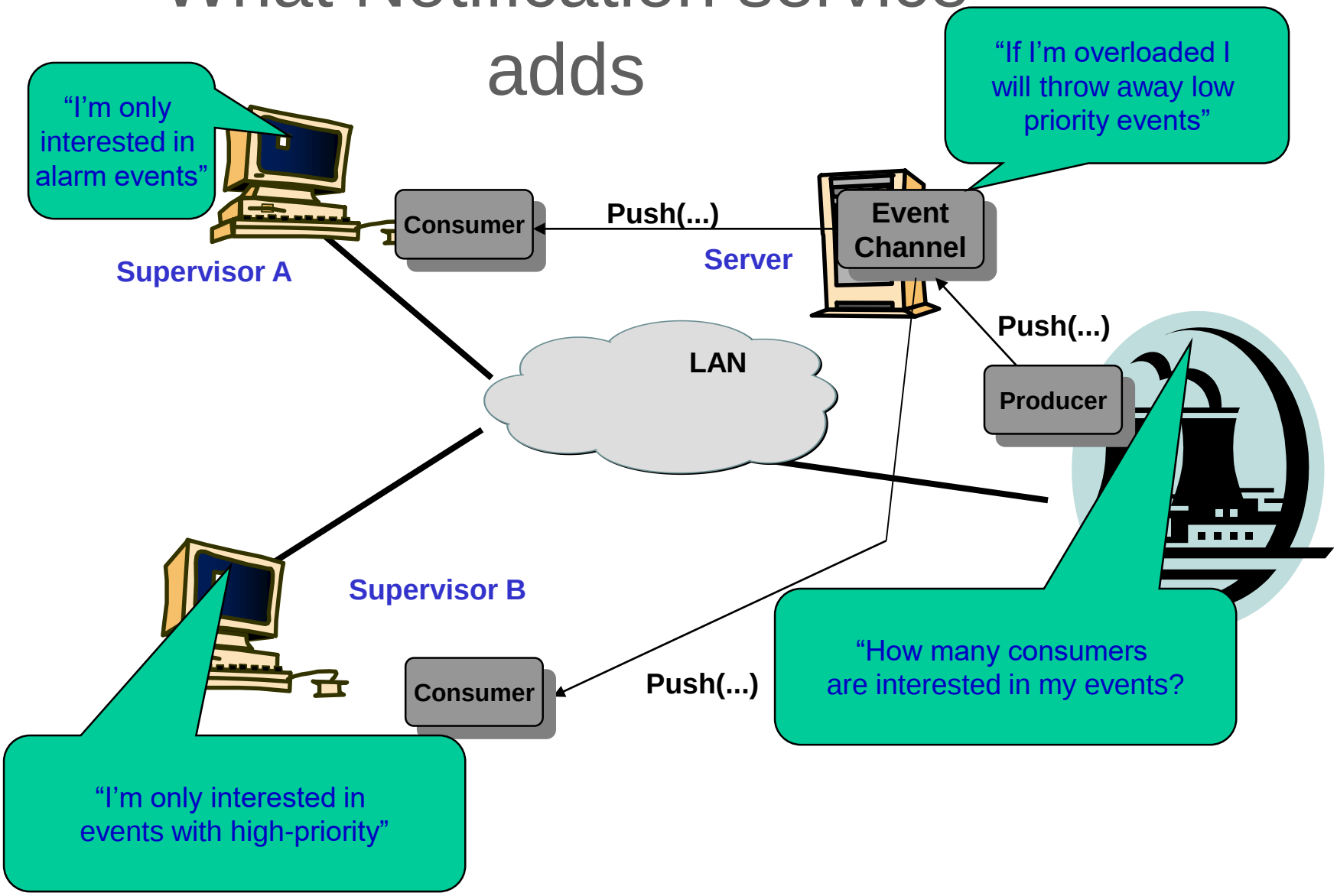
```
class PushConsumer_impl extends _PushConsumerImplBase
{
  // Implementation of push method
  public void push(Any any)
    {
        //
        // Display event data if Any is a string
        //
        try
        {
            String s = any.extract_string();
            System.out.println(s);
        }

        // What if any is not a string??
```

Limitations of EventService

- No strong data typing
- All consumers connected to an event channel receive ***all*** events
  - what if I'm only interested in events of a particular format or content?
- Event channel provides best-effort service
  - what if a high-priority event happens?
  - what if a consumer is connected at a later stage?
  - what is the channel crashes?
- A producer cannot find out who is consuming
  - how many consumers are listening to my events

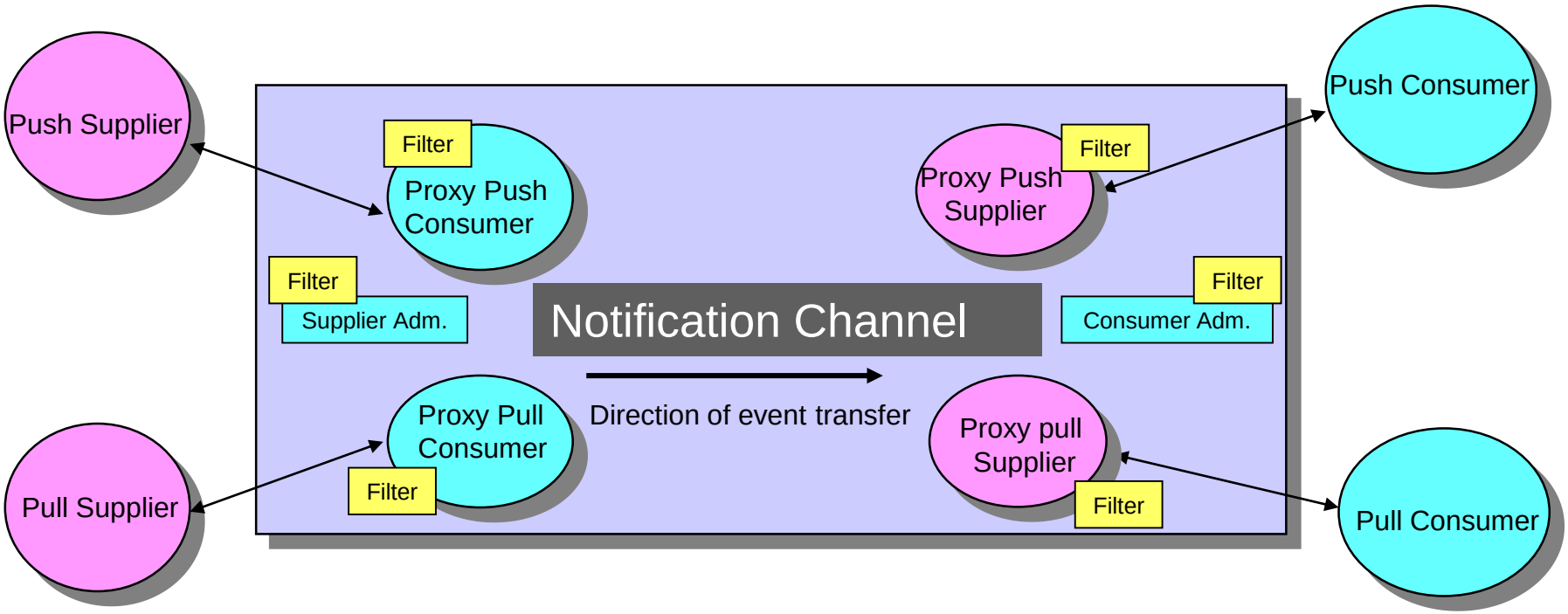
What Notification service adds



Notification Service Features

- Notification extends the OMG Event Service, adding the following features
  - Structured events
  - Event filtering using filter objects
  - QoS configurability
  - Event subscription information sharing between channels and clients
  - An optional event type repository

Notification Channel



CORBA Notification Service?

CORBA-based Notification Service is a service which extends the existing OMG Event Service, adding to it the following new capabilities:

- The ability to transmit events in the form of a well-defined data structure, in addition to Anys and Typed-events as supported by the existing Event Service.
- The ability for clients to specify exactly which events they are interested in receiving, by attaching filters to each proxy in a channel (not yet implemented in acsnc).
- The ability for the event types required by all consumers of a channel to be discovered by suppliers of that channel, so that suppliers can produce events on demand, or avoid transmitting events in which no consumers have interest.

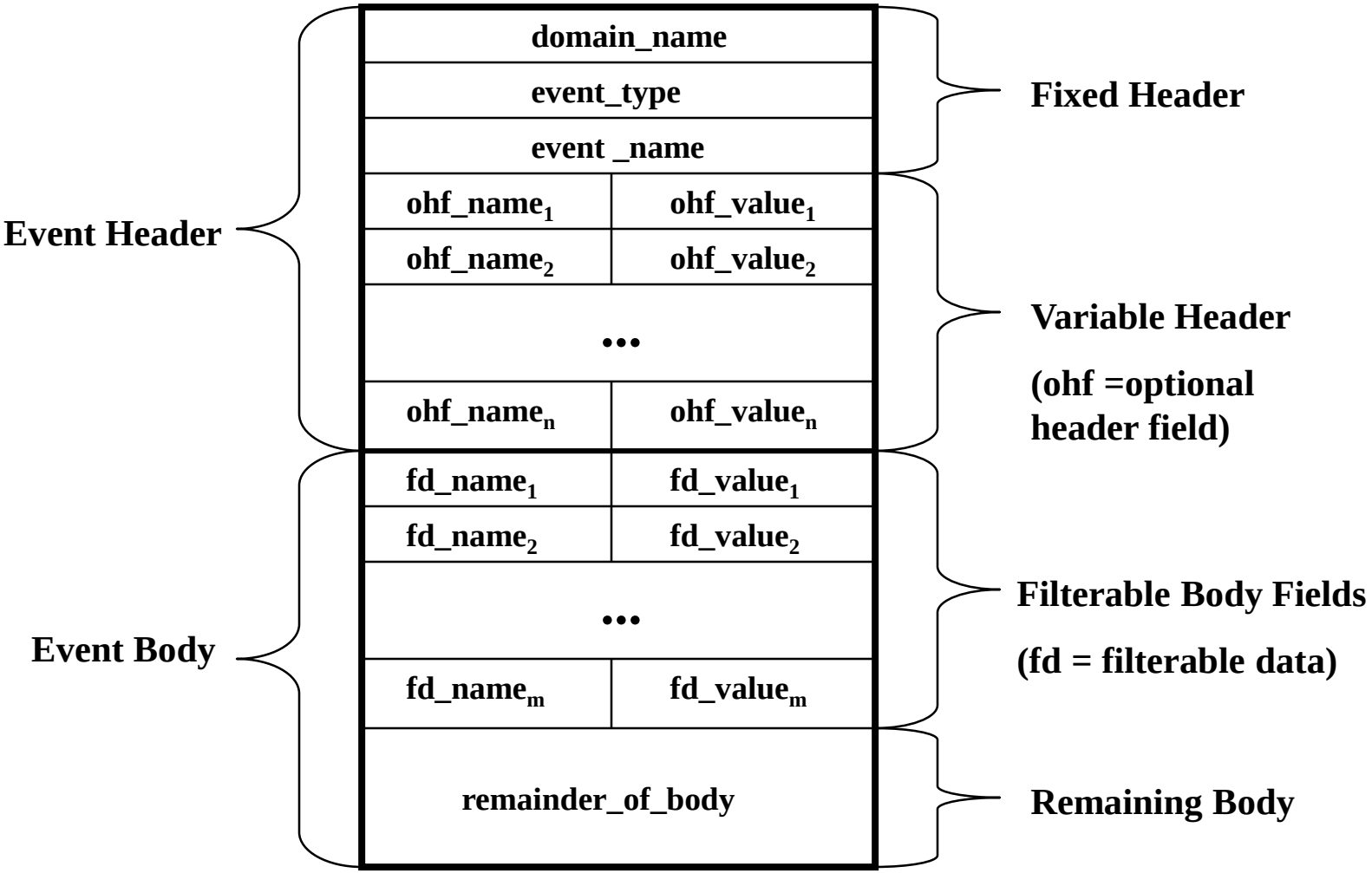
CORBA Notification Service

- The ability for the event types offered by suppliers to an event channel to be discovered by consumers of that channel so that consumers may subscribe to new event types as they become available.
- The ability to configure various quality of service properties on a per-channel, perproxy, or per-event basis.
- An optional event type repository which, if present, facilitates the formation of filter constraints by end-users, by making information about the structure of events which will flow through the channel readily available.

Structured Event

- A structured event is a data structure defined in IDL that is sent across the Notification Service by suppliers to consumers.
- There is only one field that must be “filled-out”: *type_name*. Consumers subscribe events based on this field.
- The optional fields:
  - *event_name* is basically an arbitrary string defined by the developer.
  - **Of particular interest to the developer**: *filterable_data*[]. This is a sequence of CORBA properties (string/CORBA Any pairs) that can be filtered using the extended trader constraint language discussed above.

Structure of a Structured Event



Event Filtering

Filtering is provided on three different levels:

- A consumer of structured events from “ChannelA” will never see structured events published by “ChannelB”.
- Consumers will only process structured events for the *type_name’s* it is subscribed to (which can also be dynamically unsubscribed from).
- The ability to “ignore” entire events based on various values in the structured event. A consumer provides the *type_name* and a filter string based on extended trader constraint language. This string can be as simple as “$Temperature<=100” for simple CORBA types (implying there is a CORBA integer in the structured event and it is named “Temperature”). This becomes non-trivial for filtering user-defined IDL structs though…

Consumer's Capabilities

- Subscribe to and unsubscribe from all types of events.
- Filter out structured events they don't want to process.
- The consumer doesn't have to do anything with the event's data. Can literally be used as a notification mechanism.
- Specify when they are ready to start receiving events.
- Suspend and resume their connections to the channel at any time.
- Notified when a Supplier begins publishing a new type of event and dynamically subscribe to it. The same holds true when subscriptions are no longer offered.

What is the Supplier class?

- Responsible for creating notification channels.
  - Developer can specify quality of service and admin properties if the defaults are insufficient.
  - The quality of service properties specifies things like event start time and event priority, while admin properties deal more with remote CORBA objects used in the Notification Service.

Supplier's Capabilities

- Knows when a consumer has subscribed to an event type on the channel it publishes structured events to. A "smart" supplier will only publish events (thereby reducing network traffic) when consumers are subscribed. Only useful in a one-to-many model.
- Suppliers can automatically execute a method if the connection is ever lost.
- Suppliers can destroy a notification channel.

Quality of Service Properties

- *EventReliability* - Handles the kind of guarantee that can be given for a specific event getting delivered to a *Consumer*. Represented by *BestEffort* or *Persistent*.
- *ConnectionReliability* - Handles the kind of guarantee that can be given for the connections between the Notification Channel and its clients. Can have the same values as *EventReliability.*
- *StopTime* - Specifies an absolute expiry date.
- *Timeout* - Specifies a relative expiry date.
- *StartTime* - Specifies an absolute time after which Channel starts to deliver events.
- *PriorityOrder* - Used to control the order of the events delivered to the consumers. Default value is 0 but ranges from –32767 to 32767.

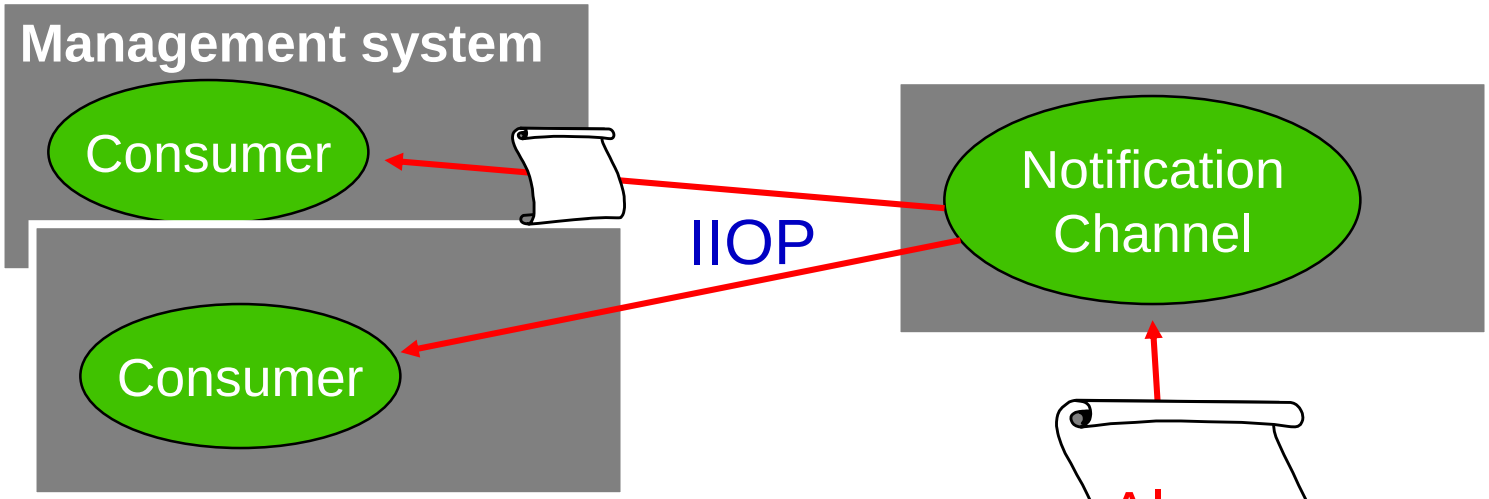
Quality of Service Properties

- *OrderPolicy* - Used by a proxy to arrange events in the dispatch queue. Can be the following values: *AnyOrder*, *FifoOrder*, *PriorityOrder*, and *DeadlineOrder*.
- *DiscardPolicy* - Defines the order in which events are discarded when overflow of internal buffers occur. Holds the same values as *OrderPolicy* plus an addition value: *LifoOrder*.
- *MaximumEventsPerConsumer* - Defines a bound for the maximum number of events the channel will queue for a given consumer. The default value is 0.
- *MaximumBatchSize* – Max. No. of events per batch
- *PacingInterval* – Max time delay between events

Administrative Properties

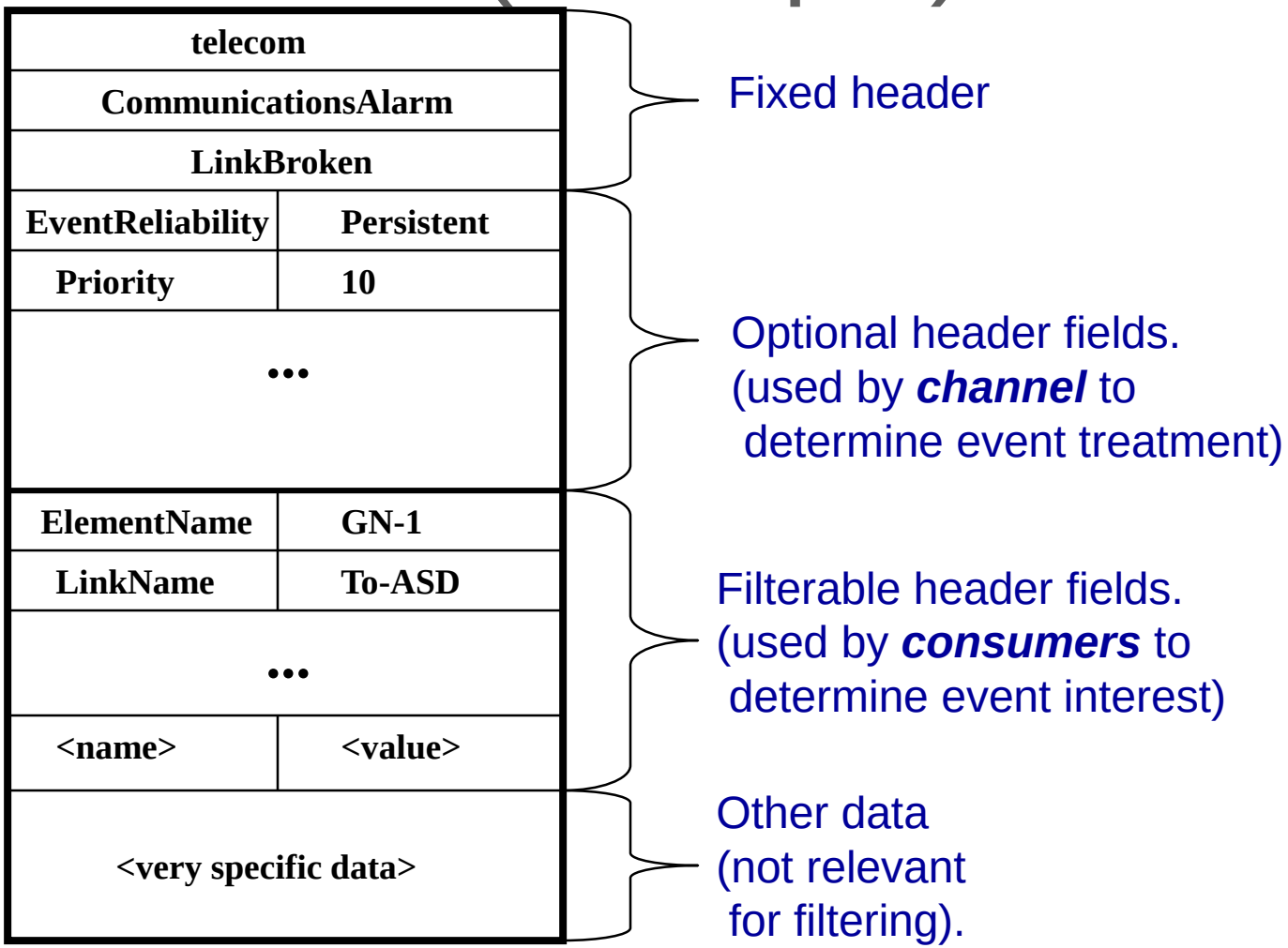
- *MaxQueueLength* - Specifies the maximum number of events the channel will queue internally before starting to discard events.
- *MaxConsumers* - Defines the maximum number of consumers that can be connected to a particular channel.
- *MaxSuppliers* - Defines the maximum number of suppliers that can be connected to a particular channel.
- *RejectNewEvents* - Defines how to handle events when the number of events exceeds *MaxQueueLength* value. Set this *Boolean* value to *true* for *IMPL_LIMIT* exceptions to be thrown. A *false* value implies events will be discarded silently.

A CORBA based Alarm notification



- Managed System acts as Supplier
- Alarm is defined as a structured event
- Consumers see events that match their interest

Alarm event (example)



AMQP 1.0 Functionality
Advanced Message Queuing Protocol

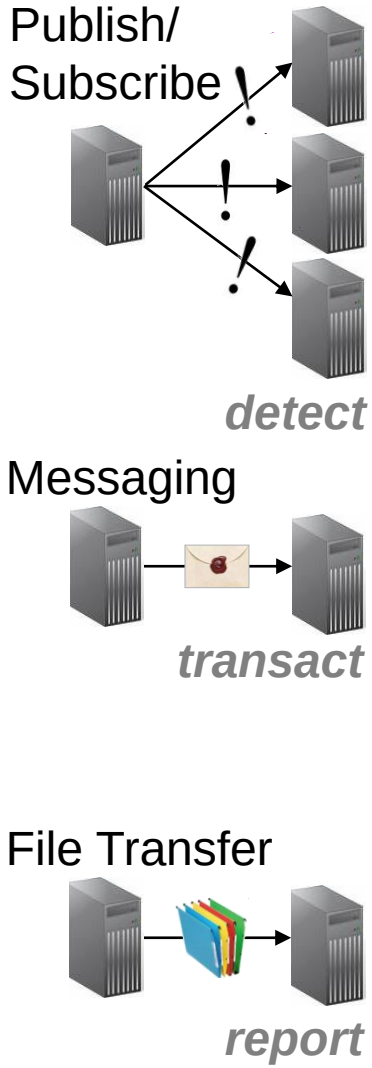
The AMQP Working Group

- Set up by JPMorgan in 2006
  - Goal to make Message Oriented Middleware pervasive
  - Make it practical, useful, interoperable
  - Bring together users and vendors to solve the problem
- We say AMQP is an "**Internet Protocol for Business Messaging**" so end users feel a connection to the technology.
  - AMQP aspires to define MOM

Cisco Systems
Credit Suisse
Deutsche Börse Systems
Envoy Technologies
Goldman Sachs
iMatix
IONA (a Progress company)
JPMorgan Chase
Microsoft
Novell
Rabbit Technologies
Red Hat
Solace Systems
Tervela
TWIST
WSO2
29West

AMQP 1.0 Covers…

- Queuing with strong Delivery Assurances
- Event distribution with Flexible Routing
- Large Message capability (gigabytes)
- Global Addressing Scheme (email-like)
- Meet common requirements of mission-critical systems

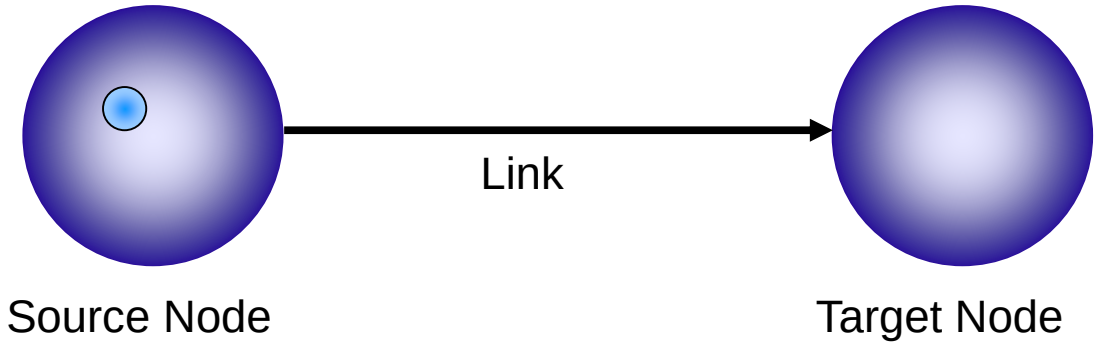


The AMQP Network

An AMQP Network consists of **Nodes** and **Links**.

A **Node** is a named source and/or sink of **Messages**.

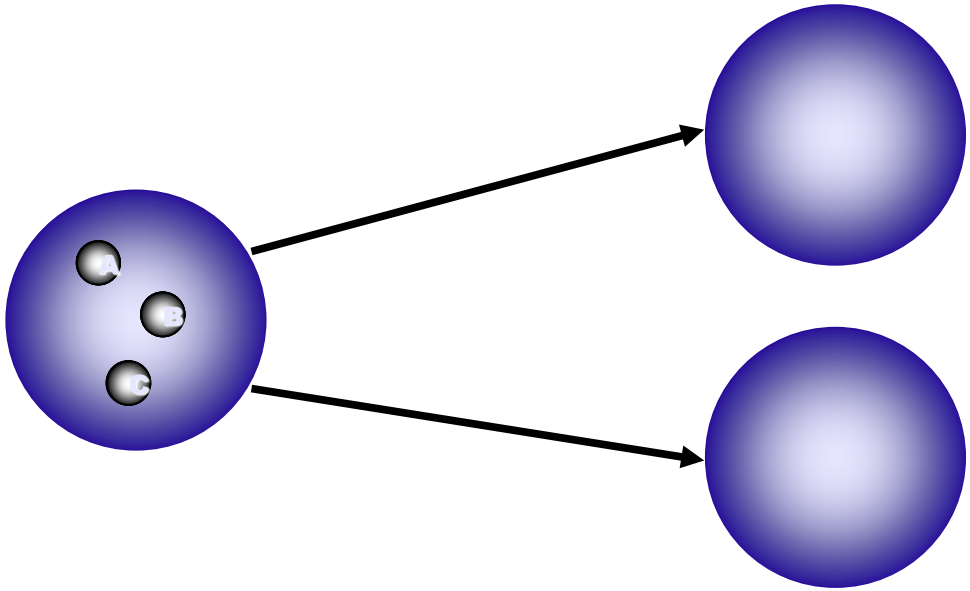
Messages travel between **Nodes** along named, unidirectional **Links.**



Types of Links

Destructive: the transfer along the link removes the message from the source

Non- Destructive: the message remains at the source node, and is “copied” to the destination.



Messages

Messages consist of parseable **Properties** and an opaque **Body**.

Messages may not be mutated by the AMQP Network.

The network carries information about the Message in **Headers** and **Footers**.

Header and Footer values may be modified in the Network.

Message Identity

A Message is assigned a globally unique identifier.

Nodes which perform transformations are creating new Messages, with new ids.

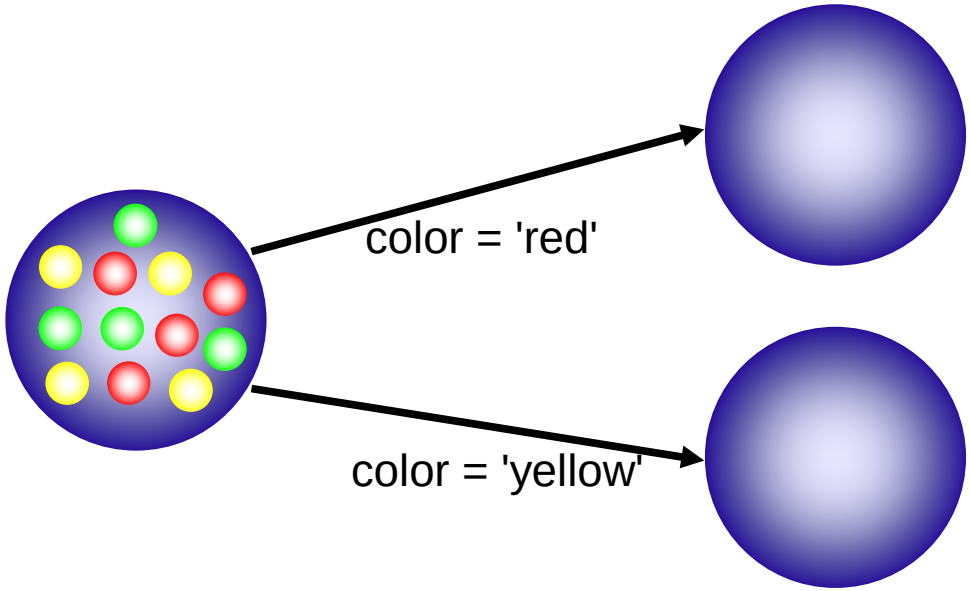
Only one “copy” of a Message can ever exist at a Node.

Filters

Each **Link** may have a **Filter** which evaluates which messages may travel along it.

Filters are evaluated against immutable properties of the Message.

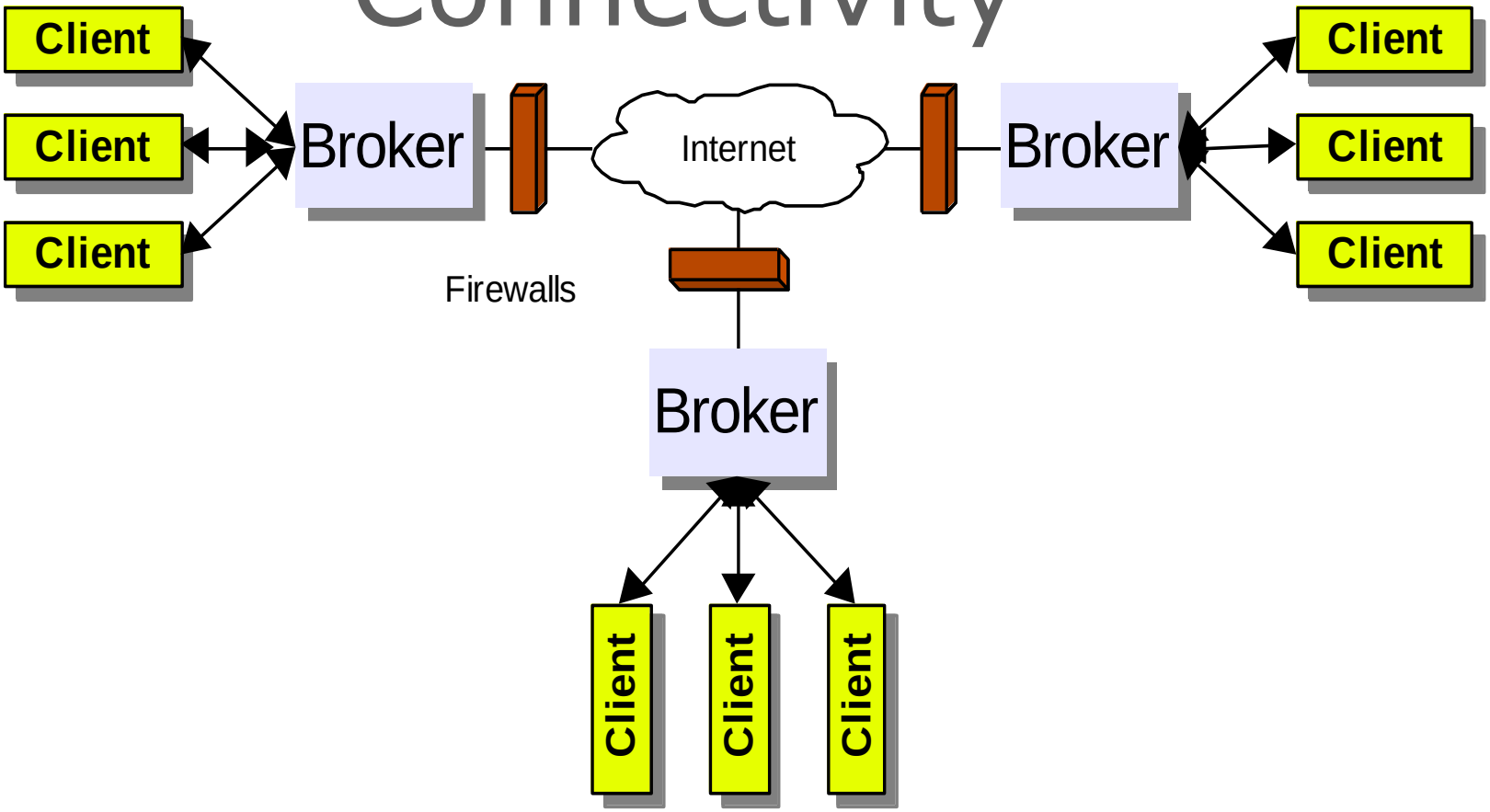
Filter syntax determined by the Filter Type, e.g. SQL, XQuery



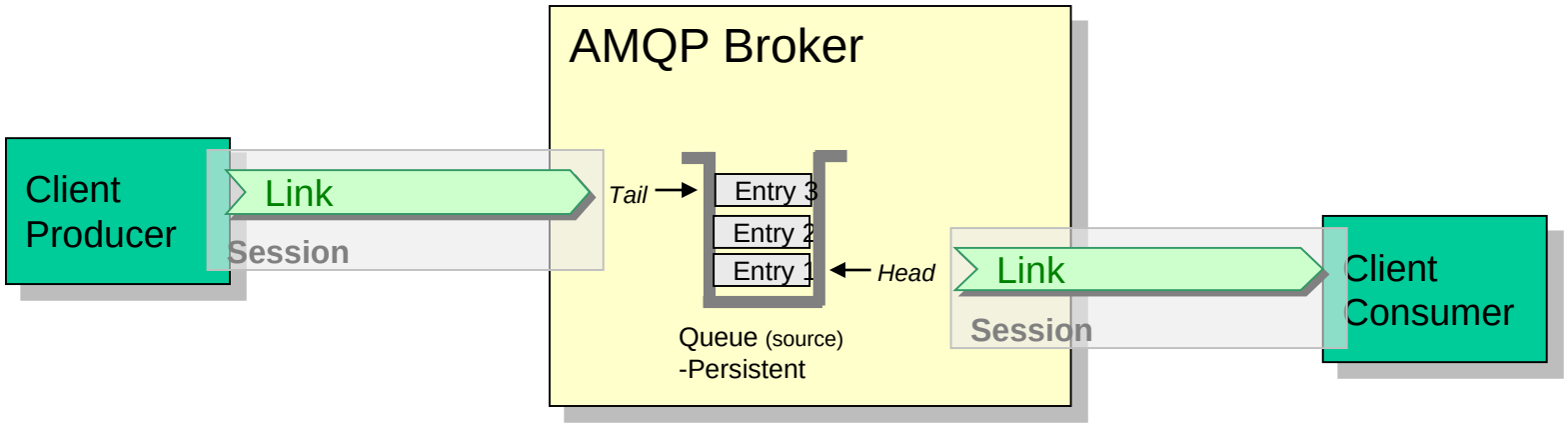
AMQP 1.0 is an Overlay Network

- **Broker**
  - Applications Connect to a Broker to participate in the AMQP network
  - The Connection is used to establish a Session
    - Sessions provide state between Connections, establish identity, ease failover
  - Connections are further subdivide into Channels
    - Multiple threads of control within an Application can share one Connection
- **Queues**
  - Applications logic interacts ONLY with Queues
  - Queues have well known Names == Addressable
  - Applications do not need to know how messages get in/out of Queues
  - Queues can be smart, they are an extension point
  - Applications will assign implied semantics to Queues (e.g. “StockOrderQueue”)
- **Links**
  - Links move Messages between Queues and/or Applications
  - Contain Routing and Predicate Evaluation Logic – similar to Complex Event Processing

Inter-Network Connectivity



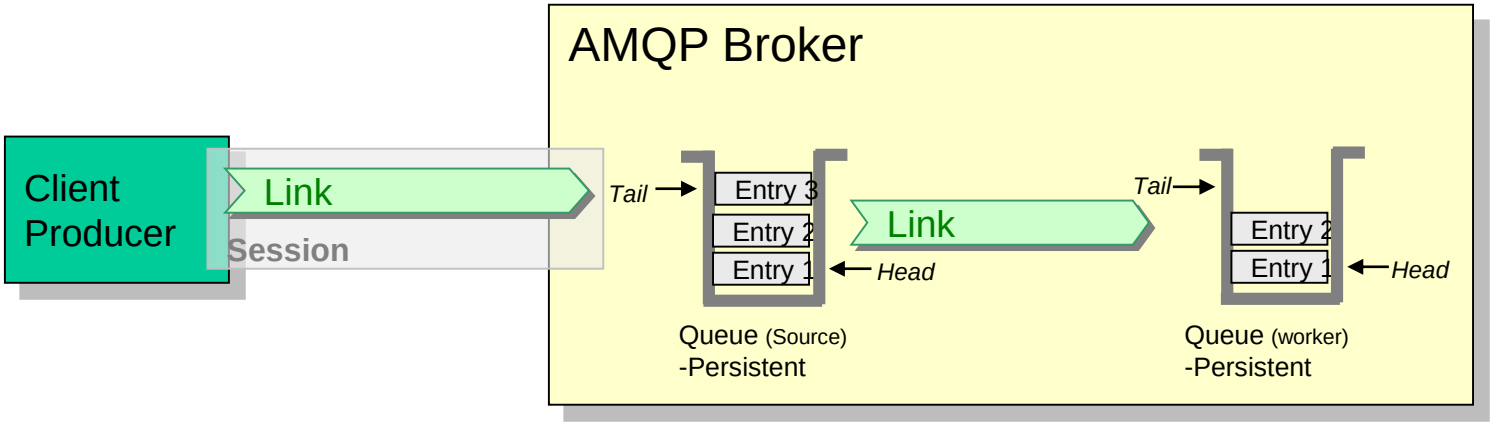
Point-to-point Queue Delivery



Highlights:

- Only “Source” queue is required and can be read directly by consumer over Link (i.e. dedicated consumer Worker queue and bridging between Source and Worker unnecessary).

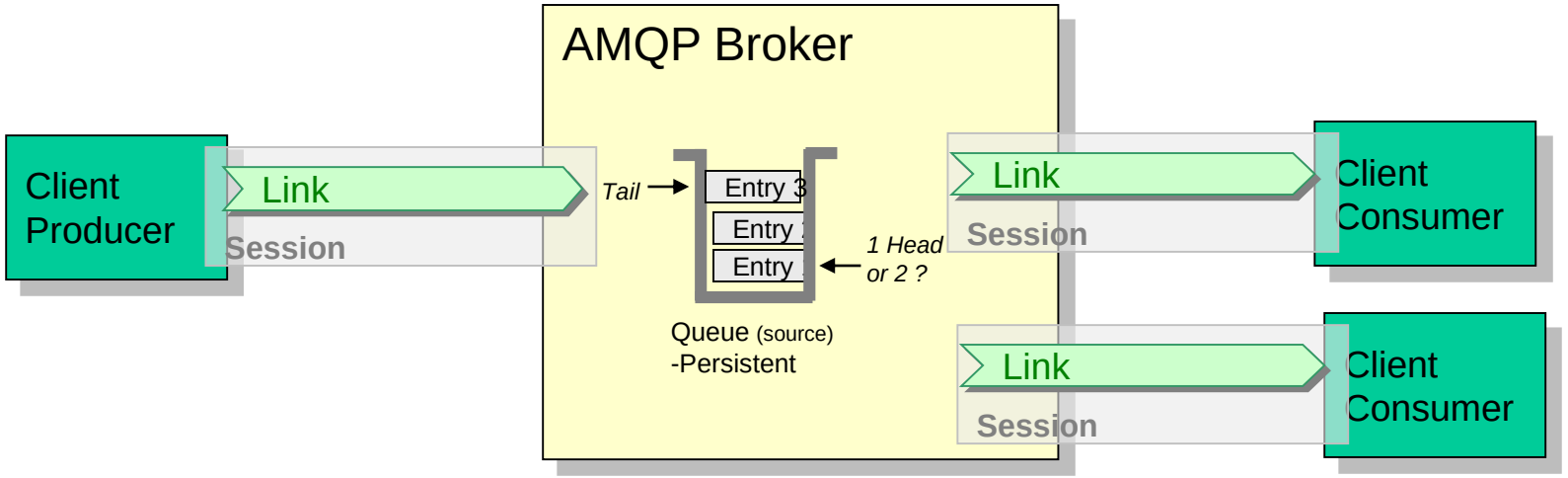
Abstracted Point-to-point Queue



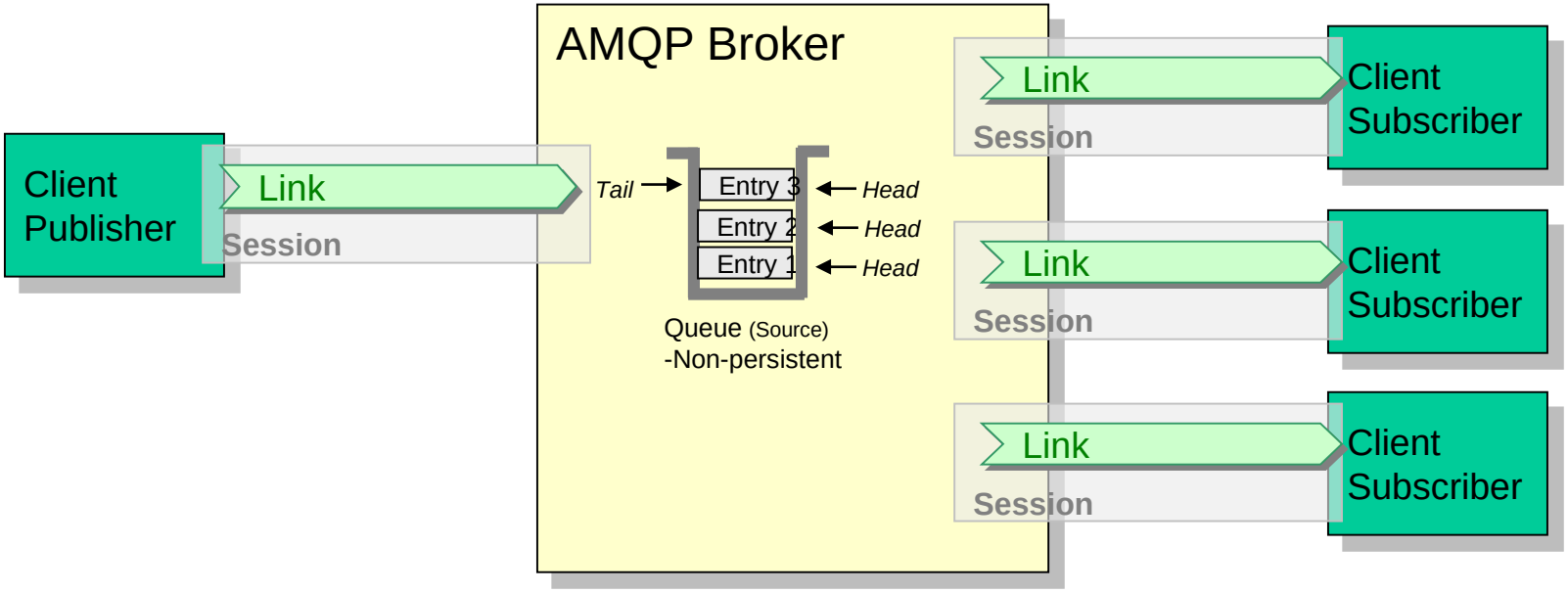
Highlights:

- One Queue performs the role of holding the “Well Known” name for the outside world.
- All messages are automatically forward on to the real worker queue.
- Allows internal topology to change without the outside world seeing (this PO Box)

Load-Balanced Point-to-point Queue Delivery



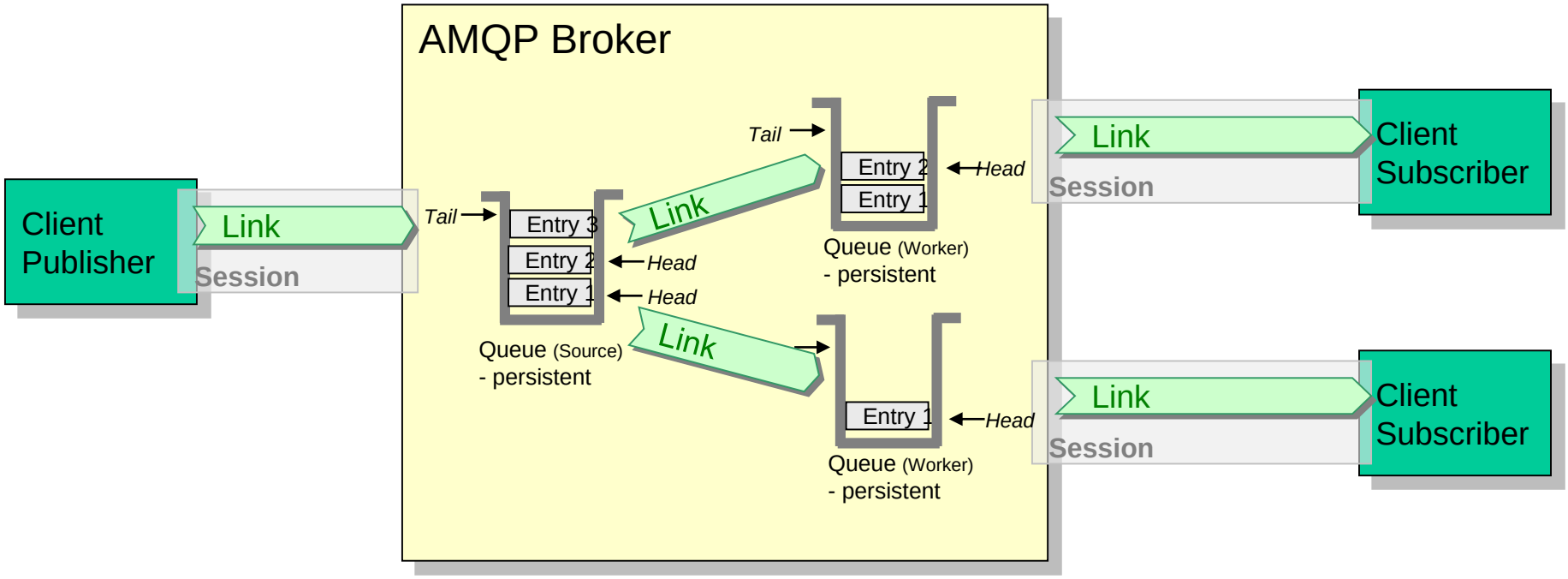
Dynamic (non-persistent) Pub/Sub Delivery



Highlights:

- Messages are “garbage collected” in an implementation specific manner after delivery.
- AMQP makes some guarantees about how long messages are valid for.

Durable (persistent) Pub/Sub Delivery



Publish-Subscribe Technology

Apache Kafka (LinkedIn)

- http://kafka.apache.org/
- Apache Kafka is a message brokering system designed for high throughput and low latency, as well as a high level of durability and fault tolerance. It uses a publish/subscribe model, where producers publish messages to a cluster of brokers, consumers poll messages from the brokers.
- Kafka was originally developed to track web site activity such as page views and searches, but it has been applied to varied activities.
- Kafka was developed at LinkedIn, and was made open source in 2011. It became a top level Apache project in 2012. Kafka uses another Apache project, Zookeeper, to maintain its broker clusters.

Apache ActiveMQ

- http://activemq.apache.org/
- Apache ActiveMQ is a very popular publish-subscribe message broker
- Mainly supports the JMS 1.1. standard. JMS is an API and not a wire protocol.
- Has its own wire protocol called OpenWire and supports standard protocols like Stomp and MQTT.
- ActiveMQ supports clustered brokers and networked brokers for scalability and fault tolerance.
- Producers and consumers can be written in different programming languages

RabbitMQ

- http://www.rabbitmq.com/
- RabbitMQ is an open source messaging framework under the Mozilla Public License.
- Developed to support the AMQP open messaging standard and has support available for other protocols like MQTT.
- Developed using erlang programming language and boasts on its high throughput and low latency.
- RabbitMQ supports clustering for fault tolerance and scalability
- Any AMQP compliant client can publish messages to RabbitMQ as well as consume messages.
- Python interface **py-librabbitmq** used by **Celery**

Apache Qpid

- https://qpid.apache.org/
- Qpid is the Apache project implementing the AMQP protocol.
- The broker is mainly written in Java and has clients written in different programming languages
- Support clustering for high availability

Kestrel

- http://robey.github.io/kestrel/
- Kestrel is a simple distributed messaging queue.
- The nodes in a Kestrel cluster doesn't communicate with each other, resulting in loosely ordered queues across the cluster.
- Because of this simple design Kestrel can scale to thousands of nodes.
- The project is developed at Twitter and available in Github.
- Kestrel supports memcache protocol and thrift based protocol for sending and receiving messages

ZeroMQ

- http://zeromq.org/
- ZeroMQ is a embeddable library for creating custom messaging solutions for applications
- Provides sockets that can be used to do inter-process, intra-process, TCP multicast messaging.
- The sockets can be connected 1 to 1, N to N with patterns like fan-out, pub-sub etc.
- The library is asynchronous in nature and provide very efficient and fast communication channels to the applications.
- Primarily developed in C but the library can be used from difference programming languages like Java, PHP etc.

Netty

- http://netty.io/
- Netty is a NIO based Java framework which enables easy development of high performance network applications like protocol Servers and Clients.
- Netty was developed at Red Hat JBoss and now available in Github under the Apache license version 2.0.
- The library provides out of the box support for popular application protocols like HTTP.
- The library can be used to build custom transport protocols using TCP or UDP.

Public Cloud Pub/Sub

- **Google Cloud Pub Sub** https://developers.google.com/pubsub is a publish-subscribe messaging system offered by Google as a cloud service
  - Supports many to many, one to many and many to one communications.
  - The publishers can use the HTTP API for sending the data.
  - The subscribers can use a pull based API or push based API for receiving the data.
  - The service is available as a developer preview and free of charge.
  - Part of Google Cloud Dataflow that also has FlumeJava and Google MillWheel
- See **Simple Notification Service** (Amazon SNS) http://aws.amazon.com/sns/ for Amazon equivalent and
- **Azure Queues** and Service Bus Queues (advanced functionality) http://msdn.microsoft.com/en-us/library/azure/hh767287.aspx for Azure equivalent
- **Apache Kafka** is open source capability

Comparison

Important Brokers changed since then

System Features	Amazon Simple Queue	Azure Queue	ActiveMQ	MuleMQ	Websphere MQ	Narada Brokering
AMQP compliant	No	No	No, use OpenWire and Stomp.	No	No	No
JMS compliant	No	No	Yes	Yes	Yes	Yes
Distributed broker	No	No	Yes	Yes	Yes	Yes
Delivery guarantees	Message retained in queue for 4 days	Message accessible for 7 days	Based on journaling and JDBC drivers to databases.	Disk store uses 1 file/channel, TTL purges messages	Exactly once delivery supported	Guaranteed and exactly-once
Ordering guarantees	Best effort, once delivery, duplicate messages exist	No ordering, Message returns more than once	Publisher order guarantee	Not clear.	Publisher order guarantee	Publisher- or time-order by Network Time Protocol
Access Model	SOAP, HTTP-based GET/POST	HTTP REST interfaces	Using JMS classes	JMS, Adm. API, and JNDI	Message Queue Interface, JMS	JMS, WS-Eventing
Max. Message	8 KB	8 KB	NA	NA	NA	NA
Buffering	NA	Yes	Yes	Yes.	Yes	Yes
Time decoupled delivery	Up to 4 days. Support timeouts.	Up to a max. of 7 days.	Yes	Yes	Yes	Yes
Security scheme	Based on HMAC-SHA1 signature. Support for WS-Security 1.0.	Access to queues by HMAC SHA256 signature	Authorization based on JAAS for authentication	Access control , authentication, SSL for communication	SSL, end-to-end application level data security	SSL, end-to-end application level data security, and ACLs
Support for Web Services	SOAP based interactions	REST interfaces	REST	REST	REST, SOAP interactions	WS-Eventing
Transports	HTTP/ HTTPS, SSL	HTTP/ HTTPS	TCP, UDP, SSL, HTTP/S, Multicast, in-VM, JXTA	Mule ESB supports TCP, UDP, RMI, SSL, SMTP and FTP	TCP, UDP, Multicast, SSL, HTTP/S	TCP, Parallel TCP, UDP, Multicast, SSL, HTTP/S, IPSec
Subscription formats	Access is to individual queues	Access is to individual queues	JMS spec allows for SQL selectors. Also access to individual queues.	JMS spec allows for SQL selectors. Also access to individual queues.	JMS spec allows SQL selectors. Access to individual queues.	SQL Selectors, Regular expressions, <tag, value> pairs, XQuery and XPath

Messaging Standards

JMS, AMQP, Stomp, MQTT I

- http://en.wikipedia.org/wiki/Java_Message_Service These are messaging standards typically used to describe messages sent from clients to brokers (like Kafka) where they are queued and inspected by subscribers
- **JMS Java Message Service** was historically very important as first broadly used/available technology for “message oriented middleware” although commercial solutions like “**MQ Series**” http://en.wikipedia.org/wiki/IBM_WebSphere_MQ from IBM were available earlier.
- JMS is supported by many brokers like RabbitMQ that also support the protocols AMQP, Stomp, MQTT
- JMS is an **API** i.e. defines how accessed in software but does not define nature of messages so different implementations can not be mixed
- AMQP, Stomp, MQTT http://blogs.vmware.com/vfabric/2013/02/choosing-your-messaging-protocol-amqp-mqtt-or-stomp.html are **protocols** i.e. nature of messages is defined so messages in these protocols should be interoperable between implementations.

JMS, AMQP, Stomp, MQTT II

- **AMQP**, which stands for **Advanced Message Queuing Protocol**, was designed as an open replacement for existing proprietary messaging middleware. Two of the most important reasons to use AMQP are reliability and interoperability.
- http://www.amqp.org/ is an OASIS standard
- AMQP is a binary wire protocol which was designed for interoperability between different vendors. Where other protocols have failed, AMQP adoption has been strong. Companies like JP Morgan use it to process 1 billion messages a day.
- As the name implies, it provides a wide range of features related to messaging, including reliable queuing, topic-based publish-and-subscribe messaging, flexible routing, transactions, and security. AMQP exchanges route messages by topic, and also based on headers.
- There's a lot of fine-grained control possible with such a rich feature set. You can restrict access to queues, manage their depth, and more. Features like message properties, annotations and headers make it a good fit for a wide range of enterprise applications.
- This protocol was designed for reliability at the many large companies who depend on messaging to integrate applications and move data around their organization.
- In the case of RabbitMQ, there are many different language implementations and great samples available, making it a good choice for building large scale, reliable, resilient, or clustered messaging infrastructures.

JMS, AMQP, Stomp, MQTT III

- STOMP http://stomp.github.io/ is the Simple Text Oriented Messaging Protocol
- Like AQMP, STOMP provides an interoperable wire format so that STOMP clients can communicate with any STOMP message broker to provide easy and widespread messaging interoperability among many languages, platforms and brokers.
- STOMP is a very simple and easy to implement protocol, coming from the HTTP school of design; the server side may be hard to implement well, but it is very easy to write a client to get yourself connected.
- STOMP does not, however, deal in queues and topics—it uses a SEND semantic with a “destination” string. The broker must map onto something that it understands internally such as a topic, queue, or exchange.
- Consumers then SUBSCRIBE to those destinations. Since those destinations are not mandated in the specification, different brokers may support different types of destination. So, it’s not always straightforward to port code between brokers.

JMS, AMQP, Stomp, MQTT IV

- MQTT (Message Queue Telemetry Transport) http://mqtt.org/ is a machine-to-machine (M2M)/"Internet of Things" connectivity protocol standardized in OASIS
- It was originally developed out of IBM's pervasive computing team and their work with partners in the industrial sector. Over the past couple of years the protocol has been moved into the open source community, seen significant growth in popularity as mobile applications have taken off.
- The design principles and aims of MQTT are much more simple and focused than those of AMQP—it provides publish-and-subscribe messaging (no queues, in spite of the name) and was specifically designed for resource-constrained devices and low bandwidth, high latency networks such as dial up lines and satellite links, for example. Basically, it can be used effectively in embedded systems.
- One of the advantages MQTT has over more full-featured "enterprise messaging" brokers is that its intentionally low footprint makes it ideal for today's mobile and developing "Internet of Things" style applications. In fact, companies like Facebook are using it as part of their mobile applications because it has such a low power draw and is light on network bandwidth.
- Some of the MQTT-based brokers support many thousands of concurrent device connections. It offers three qualities of service: 1) fire-and-forget / unreliable,2) "at least once" to ensure it is sent a minimum of one time (but might be sent more than one time), and 3) "exactly once".
- MQTT's strengths are simplicity (just five API methods), a compact binary packet payload (no message properties, compressed headers, much less verbose than something text-based like HTTP), and it makes a good fit for simple push messaging scenarios such as temperature updates, stock price tickers, oil pressure feeds or mobile notifications. It is also very useful for connecting machines together, such as connecting an Arduino device to a web service with MQTT.

The end