Middleware and Distributed Systems

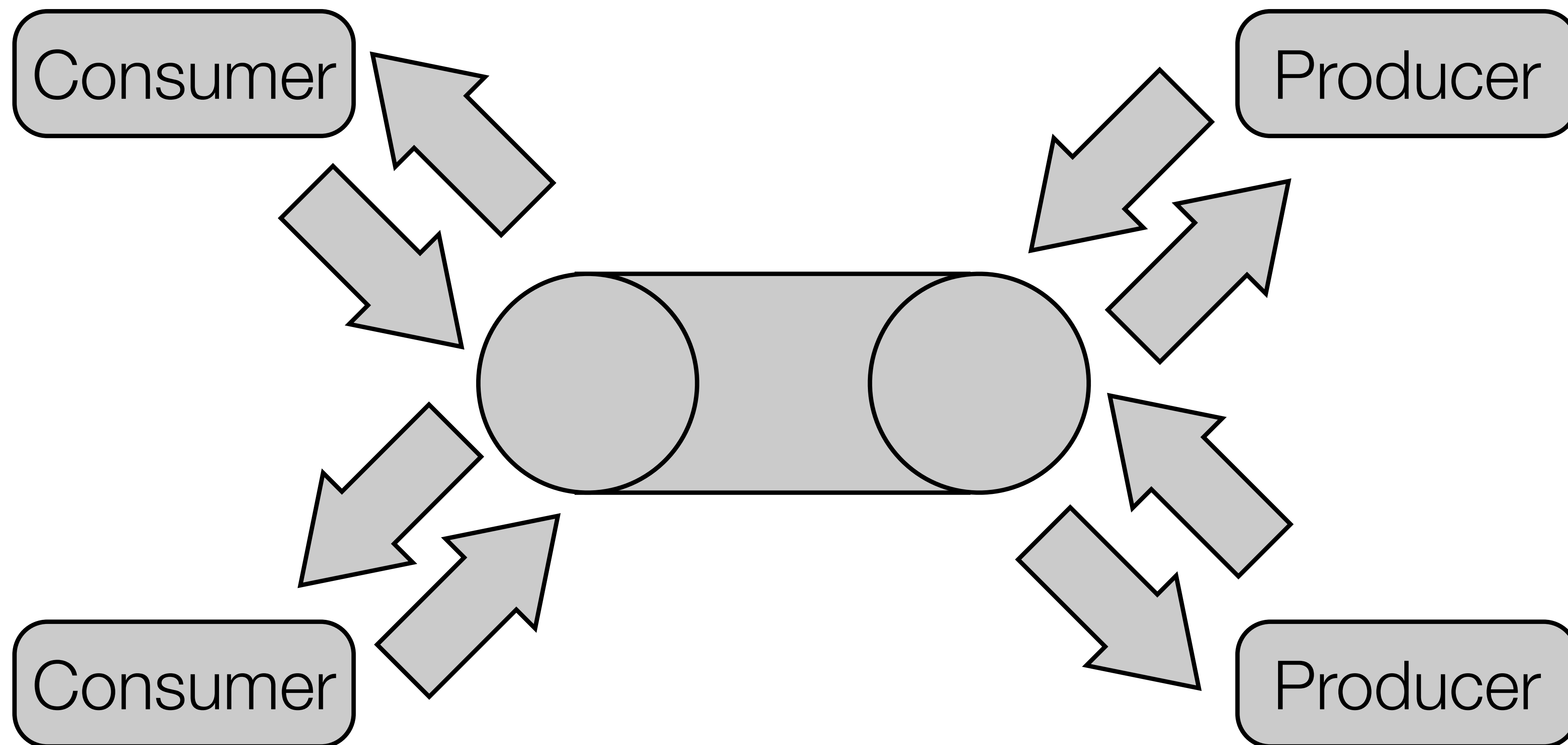
Message-Oriented Middleware

Martin v. Löwis

Message-Oriented Middleware

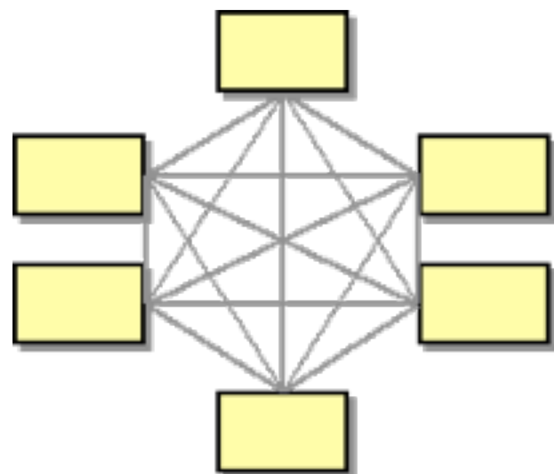
- Middleware for communication of messages between clients
- Focus on non-blocking communication style
  - Producer gives message to middleware
  - Consumer gets message from middleware
- Application responsibility for message structure, transparent to middleware
- Reliability through store-and-forward principle
- Loose coupling - decoupling of communication in space, in time, syntactically and semantically

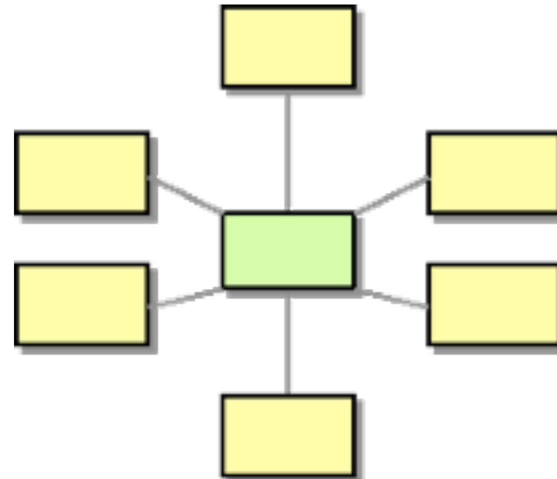
Push / Pull Model

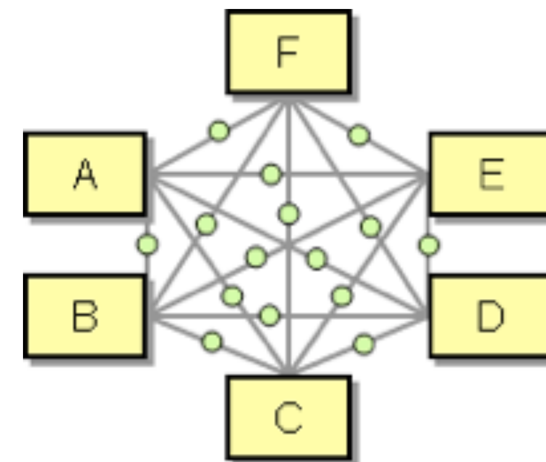


Message Brokers

- MOM provider can transform / alter the format of the message content
- ‘Hub and spoke’ architecture
  - Completely connected graph of nodes - n/2*(n-1) edges (even more with directed graph)
  - Decoupling of sender and receiver through hub (think of FedEx)
- Protocol translation and data translation
  - Common meta-format, otherwise same scalability problem for translators







(C) www.enterpriseintegrationpatterns.com

Messaging Models

- Point-to-point
  - Producing client sends message to consuming client through queue
  - Possibility of multiple producers
  - Each message delivered only once to one of the consumers
- Publish / Subscribe
  - For one-to-many or many-to-many distribution of messages
  - Subscription to topic / channel, no restriction on client role
  - All subscribed clients receive the message, if available

Persistent Message Queues

- Message queue
  - Persists message until delivery
  - Referred to using a logical name, managed by queue manager
  - Many attributes: message length, max depth, persistence, ...
- Concept of persistent queues originates from TP monitors [Bernstein90]
  - Reliable management of transaction requests
  - Recoverable queue (‘stable memory of elements’) for client / server
  - Queue repository, queue manager

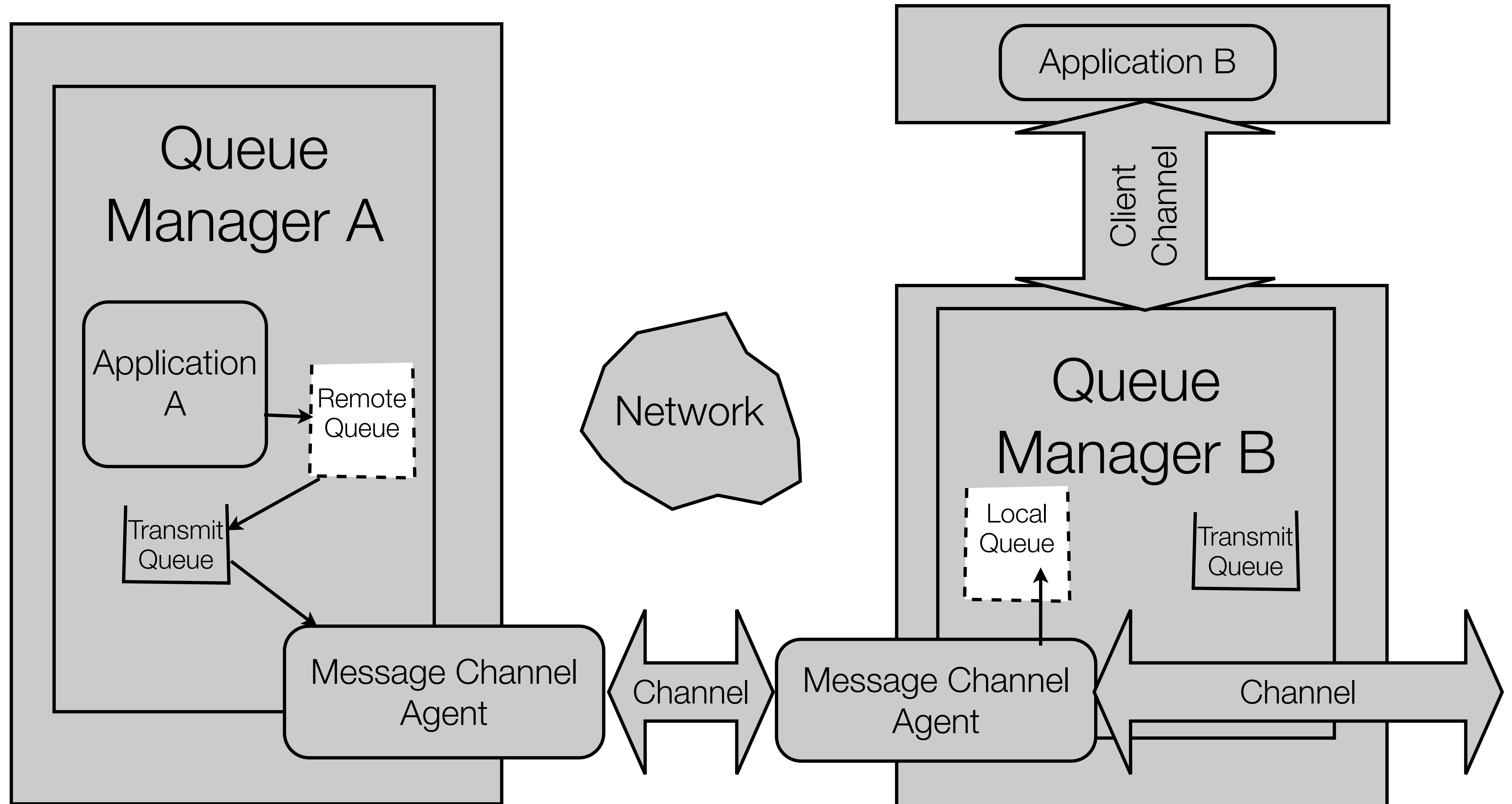
Queue Formats

- FIFO queues
- FIFO queues with priorities
- Public queue: Open access for all clients
- Private queue: Requires client authentication
- Temporary queue: Created for finite period of time or duration of a condition
- Journal queue: System keeps a copy of every message
- Connector / bridge queue: Acts as proxy between proprietary MOM systems
- Dead-Letter / Dead-Message queue: Final sink for undeliverable or expired messages

Message Delivery

- Configuration of message delivery options per queue
  - at-most-once, at-least-once, once-and-once-only delivery
  - Time-to-live (TTL) for messages
- Guaranteed message delivery
  - MOM saves pending messages on persistent storage
  - Consumer must confirm message receiving
- Certified message delivery
  - Consumption report sent to the producer

IBM MQSeries Example



Message Filtering

- Consumer / receiver demands selection of messages
- Typical filtering on properties of the message, sometimes on payload
- Boolean expression, SQL WHERE clause syntax
- Different filter types
    - Topic / Channel-based
    - Subject-based: Message subject as general property, string matching
    - Content-based: Parsing and check of payload information („stock_symbol=SUN“)
    - Pattern-based: Inspection of content over different messages („stock_symbol=SUN AND ms_price < 50$“)

Transactional Messaging

- Group tasks into a single unit of work
    - All message delivery is completed, or all will fail together
    - Messages are not forwarded to the consumer until transaction commit by the sending producer
    - Messages are not removed from the MOM until transaction commit by the receiving consumer
    - Typically represented by transactional queue
- Local transactions within a single resource manager
- Global transactions with multiple (heterogeneous) distributed resource managers, and external transaction coordinator
- Queued Transaction Processing (QTP): Queues as transactional resources

Commercial Products

- TIBCO Rendezvous
- IBM Websphere MQ (MQSeries)
- Sun ONE Messaging Server
- Microsoft Message Queue Server (MSMQ)
- Sonic MQ -> ESB
- Common focus on:
  - Guaranteed message delivery features
  - Scalability regarding message throughput
- Few standardization attempts (AMQP)

CORBA Event Service

- Alternative to RPC-based client-server communication
- CORBA event service: Standardized interfaces from OMG (1995 - 2004)
  - Extension with CORBA notification service
- Event: “something that happens”, event consumer and event supplier
- Notification: Message that informs about some kind of event
- Push (active producer) and pull (active consumer) model
- Single source, (possibly) multiple recipients
  - Source does not know all consumers
  - Non-blocking sending of message
  - Typed communication (untyped communication through *any* type)

Stock Exchange Example

```
interface StockExchange;
struct StockQuote
{
  string stock_id;
  StockExchange market_place;
  double current_quote;
  Time current_time;
};
interface Subscriber
{
  void receive (in ::StockQuote current_quote);
};

interface StockExchange
{
  void subscribe (in ::Subscriber customer);
};
```

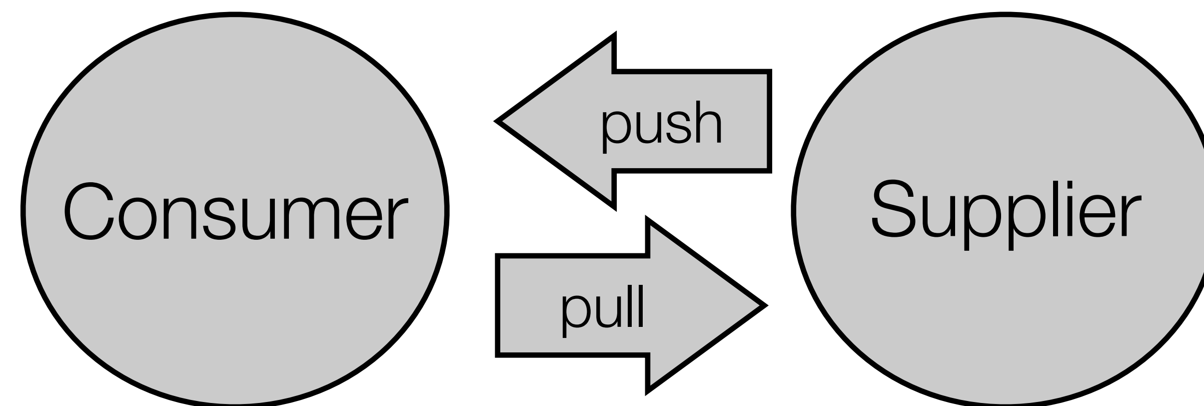
CORBA Event Service Interfaces

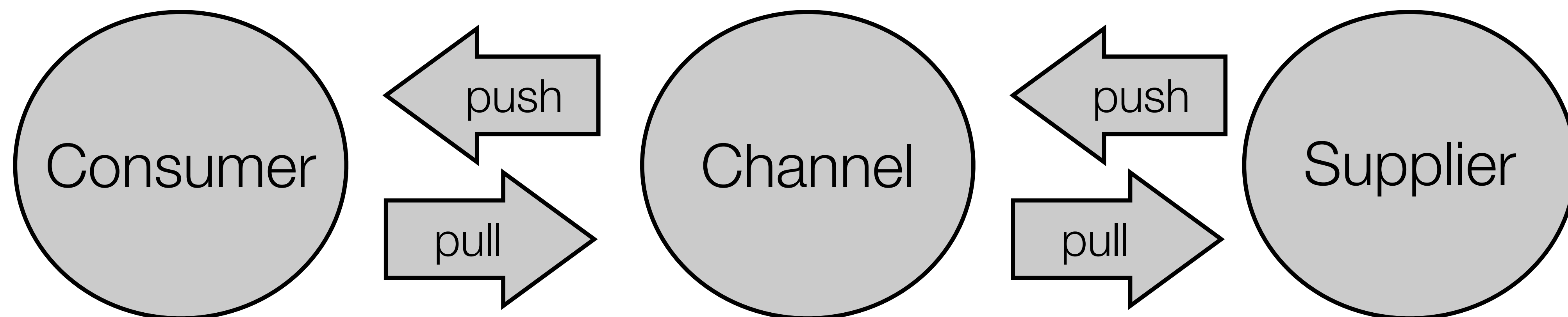
```
interface PushConsumer {
     void push (in any data) raises(Disconnected);
     void disconnect_push_consumer();
};
interface PushSupplier {
     void disconnect_push_supplier();
};



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();
};
```

Stock Exchange Example - 2nd try

```
interface StockExchange2;
struct StockQuote
{
  ...
  StockExchange2 market_place;
  ...
};
interface Subscriber2 : ::CosEventComm::PushConsumer
{};

interface StockExchange2 : ::CosEventComm::PushSupplier
{
  void subscribe (in ::Subscriber2 customer);
};
```

Event Channel as Middleman



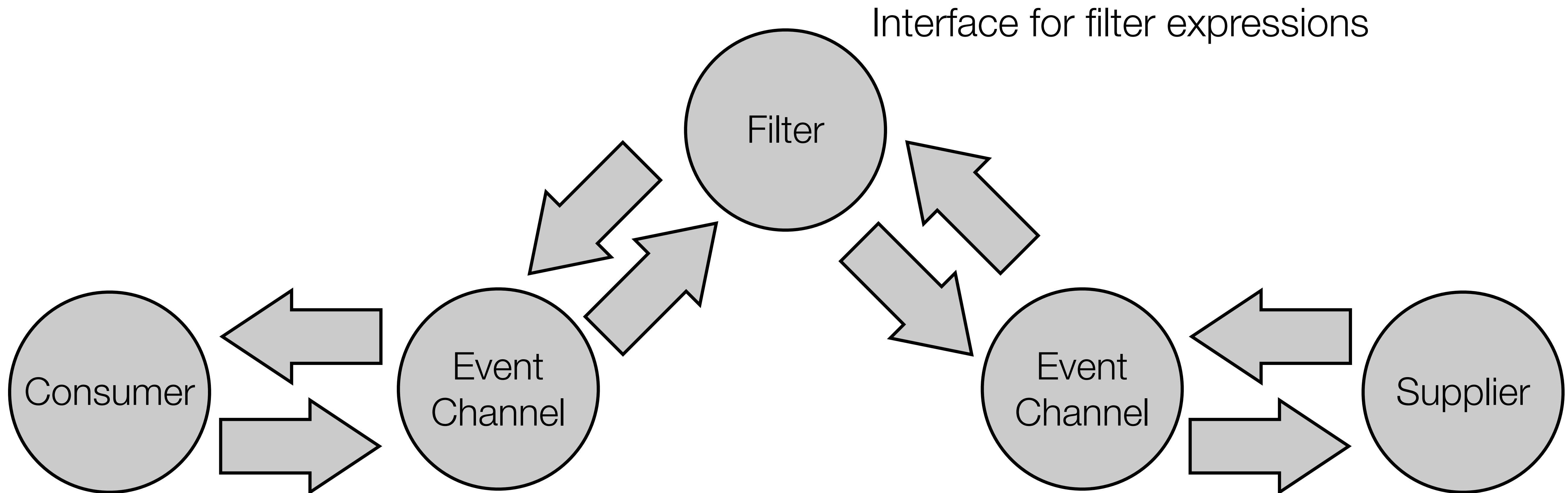


CORBA Event Channel Usage

```
interface Channel
  : ::CosEventComm::PushSupplier,
    ::CosEventComm::PushConsumer {
  void register_supplier (
      in ::CosEventComm::PushSupplier supplier);
  void register_consumer (
      in ::CosEventComm::PushConsumer consumer);
};

interface MyConsumer : ::CosEventComm::PushConsumer
{};

interface MySupplier : ::CosEventComm::PushSupplier
{};
```

CORBA Event Filters

- Subscriber might be only interested in specific notifications
- Demands interface for filter installation



CORBA Notification Service

- Wide usage of Event service, several proprietary extensions
- Extended event service in CORBA 3 -> Notification Service
- Filter objects
  - Interfaces to add, remove and modify constraints on message values
- Quality of service
  - Interfaces for control over notification delivery characteristics
  - Examples: discard policy, earliest delivery time, expiration time, max events per consumer, ordering policy, event priority, re-connection
- Demands general standardized event structure

Structured Notification Event

- Header with fixed and variable part
  - domain_name: vertical industry domain (telco, finance, health care)
  - type_name: type of event within domain (StockQuote, VitalSigns)
  - event_name
  - Optional header fields, some standardized names
    - *EventReliability* / *ConnectionReliability* (short): 0 - best effort, 1 - persistent
    - *Priority* (short): -32.767 ... 0 ... 32.767
    - *StartTime*, *StopTime* or *Timeout* for delivery
- Body with filterable data and raw data

CORBA Notification QoS Levels

Property	Per-Message	Per-Proxy	Per-Admin	Per-Channel
EventReliability	X			X
ConnectionReliability		X	X	X
Priority	X	X	X	X
StartTime	X			
StopTime	X			
Timeout	X	X	X	X
StartTimeSupported		X		X
StopTimeSupported		X		X
MaxEventsPerConsumer[1]		X		
OrderPolicy		X	X	
DiscardPolicy[1]		X	X	
MaximumBatchSize[2]		X	X	X
PacingInterval[2]		X	X	X

AnyOrder
FifoOrder
PriorityOrder
DeadlineOrder

AnyOrder
FifoOrder
LifoOrder
PriorityOrder
DeadlineOrder

Structured Event IDL

```
module CosNotification {
   ...
   struct FixedEventHeader {
      _EventType event_type;
      string event_name;
   };

   struct EventHeader {
      FixedEventHeader fixed_header;
      OptionalHeaderFields variable_header;
   };

   struct StructuredEvent {
      EventHeader header;
      FilterableEventBody
           filterable_data;
      any remainder_of_body;
   }
   ...
}
```

```
struct _EventType {
   string domain_name;
   string type_name;
};
```

```
typedef string Istring;
typedef Istring PropertyName;
typedef any PropertyValue;
struct Property {
   PropertyName name;
   PropertyValue value; };
typedef sequence<Property> PropertySeq;
typedef PropertySeq OptionalHeaderFields;
typedef PropertySeq FilterableEventBody;
```

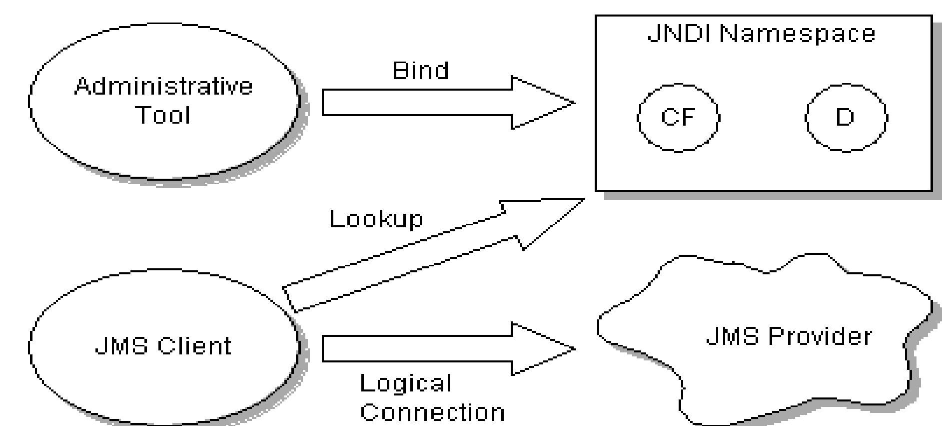
Notification Service Constraint Language

- Extended version of CORBA Trading service constraint language
  - Special token “$” for current event and any run-time variable
  - Reserved variable “$curtime”
  - Vendor extensions have “:” prefix

```
$type_name == ‘CommunicationsAlarm’ and not
($event_name == ‘lost_packet’)

$type_name == ‘CommunicationsAlarm’ and
$priority >= 1 and $priority <= 5

$origination_timestamp.high + 2 < $curtime.high

exist $threshold._type_id and $threshold._type_id == ‘short’
```

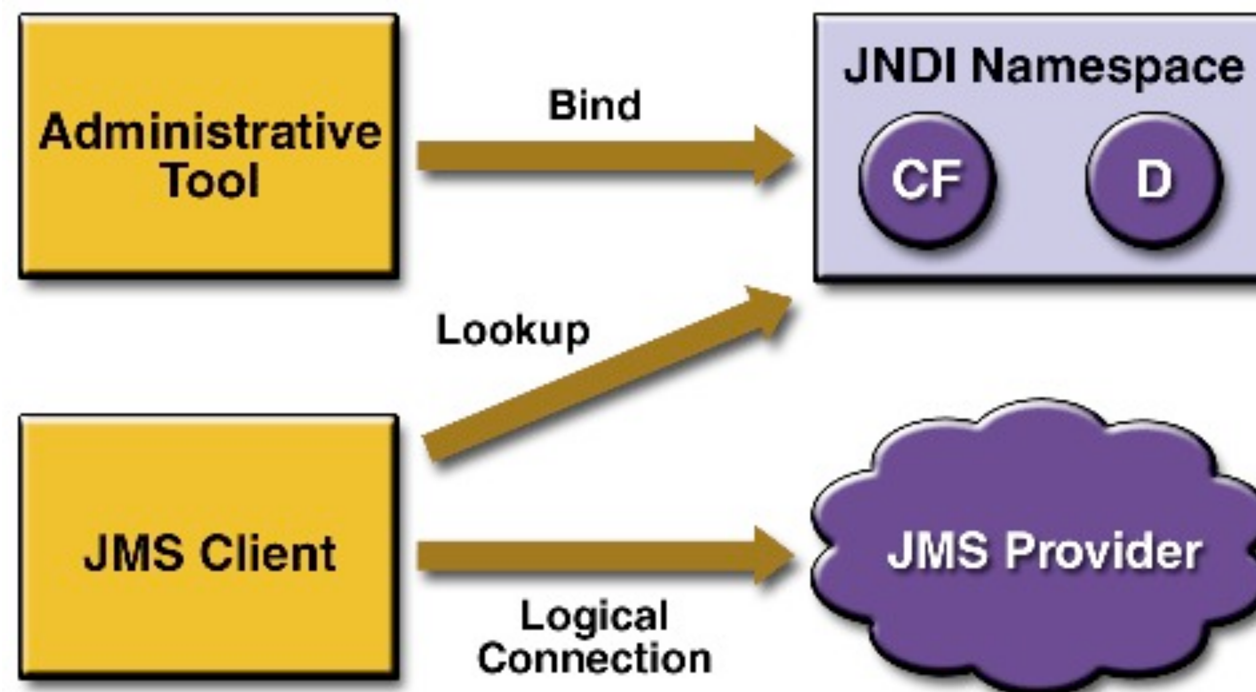
Java Messaging Service

- Client API specification for messaging in Java (JSR 914, 2001)
- „...common set of enterprise messaging concepts...“
- No wire protocol, no security, no error notification
- JNDI typically used to get administered JMS objects
  - Administrative tasks outside of client code
  - *ConnectionFactory* - used to create JMS provider connection
  - *Destination* - used for specifying the destination of messages

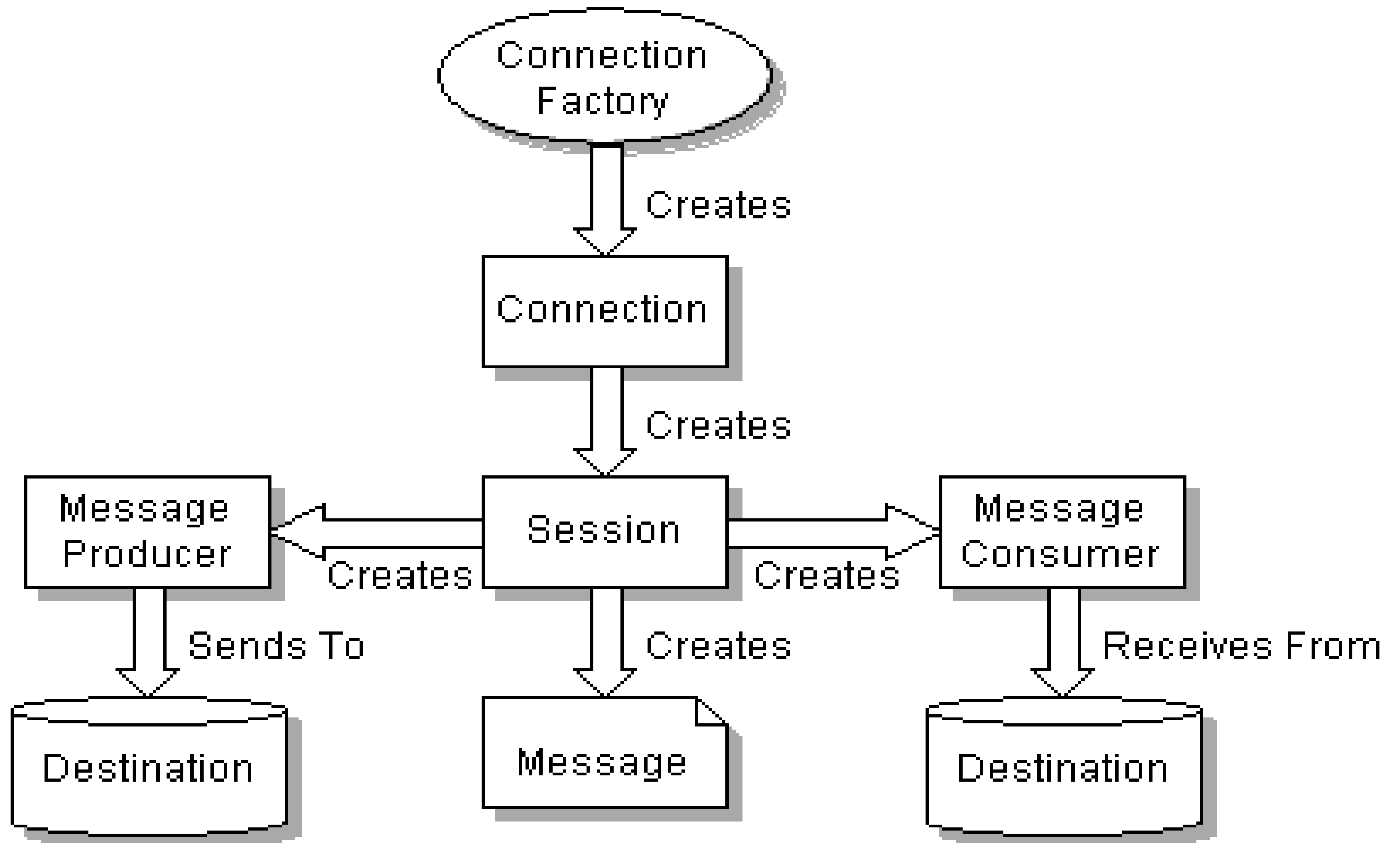


JMS Provider

- „...entity that implements JMS for a messaging product...“ - JAR file
- Producer gives message to provider for delivery
- Consumer gets message synchronously / asynchronously from provider



JMS Relationships



JMS Interface Comparison

JMS Common Interfaces	**PTP-specific Interfaces**	**Pub/Sub-specific interfaces**
ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory
Connection	QueueConnection	TopicConnection
Destination	Queue	Topic
Session	QueueSession	TopicSession
MessageProducer	QueueSender	TopicPublisher
MessageConsumer	QueueReceiver, QueueBrowser	TopicSubscriber

Developing a JMS Client

- Use JNDI to find a *ConnectionFactory*
- Use JNDI to find one or more *Destination* objects
- Use *ConnectionFactory* to establish connection with demanded delivery feature
- Use *Connection* to create a *Session*
- Create *Producer / Consumer* object through the *Session* to a particular *Destination*
  - Specific class depends on messaging model

JMS Messages

- Standardized message interfaces
- Message implementation by JMS provider
- Message header information
  - Common set of mandatory header fields for receiver
  - Facility for adding properties to header (standard, application-specific, provider-specific)
- Some predefined message body structures
  - Still no promise for interoperability !

JMS Message Header

- *JMSDestination*, *JMSDeliveryMode*
- *JMSMessageId*: Assigned by provider
  - Scope of uniqueness is implementation-specific
- *JMSTimestamp*: Time of delivery to provider
- *JMSCorrelationID*: might be used for relation of response message to request message
- *JMSRedelivered*: Indication for consumer
- *JMSType*: Reference to provider message repository, no defaults specified by JMS

JMS Message Header

- *JMSExpiration*: „...clients SHOULD not receive messages that have expired...“
- *JMSPriority*: 0-4 (normal), 5-9 (expedited)
- Administrator can override *JMSDeliveryMode*, *JMSExpiration* and *JMSPriority*

Header Fields	**Set By**
JMSDestination	Send Method
JMSDeliveryMode	Send Method
JMSExpiration	Send Method
JMSPriority	Send Method
JMSMessageID	Send Method
JMSTimestamp	Send Method
JMSCorrelationID	Client
JMSReplyTo	Client
JMSType	Client
JMSRedelivered	Provider

JMS Message Properties

- Additional set of optional header elements with “JMSX” prefix
- Set by provider on send
  - *JMSXUserId, JMSXAppId, JMSXProducerTXID*
- Set by provider on receive
  - *JMSXDeliveryCount, JMSXConsumerTXID, JMSXRecvTimestamp*
- Set by client
  - *JMSXGroupId, JMSXGroupSeq*

JMS Acknowledgment Modes

- Controlled at session level
- *AUTO_ACKNOWLEDGE*
  - Asynchronous mode: Handler acknowledges successful return (???)
  - Synchronous mode: Client has sucessfully returned from receive()
- *CLIENT_ACKNOWLEDGE*
  - Client calls acknowledge() by itself
- *DUPS_OK_ACKNOWLEDGE*
  - Lazy acknowledge which might lead to duplicates
  - Consumer must be able to handle duplicates
    -> reduces messaging overhead

JMS Message Types

- TextMessage: java.lang.String object
- MapMessage: Set of name/value pairs
- BytesMessage: Stream of bytes
- StreamMessage: Stream of Java primitive values, read sequentially
- ObjectMessage: Serializable Java object

	boolean	**byte**	**short**	**char**	**int**	**long**	**float**	**double**	**String**	**byte[]**
boolean	X								X	
byte		X	X		X	X			X	
short			X		X	X			X	
char				X					X	
int					X	X			X	
long						X			X	
float							X	X	X	
double								X	X	
String	X	X	X		X	X	X	X	X	
byte[]										X

JMS Example Code

```
import javax.naming.*;
import javax.jms.*;

Context c = new InitialContext();
ConnectionFactory cf = (ConnectionFactory) c.lookup("ConnectionFactory");
Queue stockQueue = (Queue)c.lookup("StockSource");
Connection connection = ConnectionFactory.createConnection();
Session session;                    // no TA, automated ack of messages
session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
MessageProducer sender = session.createProducer(stockQueue);
String selector = new String("(StockSector = 'Technology')");
MessageConsumer receiver = session.createConsumer(stockQueue,selector);
connection.start();

MapMessage message = session.createMapMessage();
message.setString("Name",  "SUNW");
message.setDouble("Value", stockValue);
message.setStringProperty("StockSector", "Technology");
sender.send(message);

MapMessage message = (MapMessage)receiver.receive();
```

Messaging In The Internet

MOM	Email
MOM message	SMTP message
Message queue	Mailbox
Consumer	POP3 client
Producer	SMTP client
Queue manager	MTA
Routing key	To: / Cc: address
Publish / Subscribe	Mailing list
Message filter	Server-side spam check
Message acknowledge	
Transactional messaging	
Time to live	