Fan out

- send events to all consumers

Load balance

- send event to one consumer/subscriber at random, leave it to them to communicate among themselves?

Fault tolerance

Apache Kafka

- log based streaming?

Stream processing vs batch processing

- the difference is that streams are unbounded
- a batch knows when it has finished reading its input
- but a lot of data is unbounded and never really ends?
  - users will produce data yday and today and tomorrow...
- you could split a batch processing pipeline into time based chunks -- but for many use cases you need real time processing

In general, a stream refers to data that is made incrementally available over time

- eg: stdin and stdout on unix, filesystem APIs

In the batch processing world, the inputs and outputs of a job are files, perhaps on a distributed filesystem.

- typically, the first task is the parse files into records

In the streaming world, these records are typically called events

- each event is timestamped, and contains details on some action
- could be encoded as JSON or text or a binary format

An event is generated once by a producer and then processed by possibly multiple consumers

- producer = publisher, consumer = subscriber

In a filesystem, a filename identifies a set of related records; in a streaming system, related events are usually grouped together into a topic or stream

In principle, a file or db is sufficient to connect producers and consumers

- producer writes to this data store - consumer polls data store to check for events that have appeared since it last ran -- analogous to what a batch process does when it processes a day's worth of data at the end of every day
- However, if the datastore is not optimized for it, continous processing with low delays can become expensive

    - the more often you poll, the lower the hit rate
    - instead better for consumers to be notified when new updates arrive -- but most dbs dont support these kinds of triggers

Messaging systems

- a common approach for notifying consumers about new events is to use a messaging system
    - a producer sends a message containing the event, which is then pushed to consumers
    - examples: unix pipes, TCP connection between producer and consumers
    - this is a basic model: 1 prod 1 consumer
    - want to allow > 1 producer and > 1 consumer
- This is called a publisher/subscriber model
    - and different systems take a wide range of approaches to this
- Questions to ask to understand different systems that attempt to address this
    - what happens if producers send messages faster than the consumers can process them?
        * drop messages
        * buffer in queue
        * backpressure/flow control
            · this is what happens in unix pipes and TCP -- when the small fixed size buffer is filled, the sender is blocked until buffer space clears up
    - if data buffers in a quque, what happens as the queue grows?
        * if the q doesnt fit in memory? Is it written to disk?
    - what happens if nodes crash or go online? Are messages lost?
        * durability?
        * whether message loss is acceptable depends on the application

Direct messaging from producer to consumers (without intermediary nodes)

- UDP multicast used in the financial industry where low latency is important
    - application level protocols can recover lost packets
- Brokerless messaging libraries like ZeroMQ
- if the consumer exposes a service on the network, a webhook pattern may be useful
    - the callback URL of one service is registered with another service
    - it makes a request to that URL whenever an event occurs

Message brokers

- A widely used alternative is to send mesages via a message broker or message queue
- this is a kind of db that is optimized for handling message streams
- it runs as a server, with producers and consumers connecting to it as clients
- a broker assigns individual messages to consumers, and consumers acknowledge

messages when they have been processed
- the question of durability is moved to the broker
  - some keep messages only in memory, others write to disk for durability
  - faced with slow consumers they generally allow unbounded queueing, as opposed to dropping messages or backpressure
- a consequence of queueing is that consumers are generally asynchronous
  - producer sends message to broker -- it normally only waits for broker to confirm that it has buffered the message and does not wait for consumers to process the message
  - the delivery to consumers will happen at some undetermined future point in time

Some important practical differences between databases and message brokers
- dbs usually keep data until it is explicitly deleted. most brokers automatically delete a message after it has been delivered to its consumers
  - so not suitable for long term data storage
- dbs support index-based searching. Mbs often support some way of subscribing to a subset of topics matching some pattern.
  - in essence both are ways for a client to specify some subset of the data that it wants to know about

This is a traditional view of message brokers, used in standards in JMS and AMQP
and implemented in software like RabbitMQ, ActiveMQ, HornetQ, Qpid, TIBCO Enterprise MEssage Servie, Google Cloud pub/sub

Multiple Consumers

When multiple consumers read messages in the same topic, a message broker usually does one of either

1. Load Balancing: each message is delivered to one consumer, so consumers share the work of processing the messages
2. Fan out: each message is sent to every consumer
   - the streaming equivalent of every batch job reading the same input file

A second kind of message broker: Log-based message brokers

- A log is simply an append-only sequence of records of disk
- the same structure can be used to implement a message broker
  - a producer sends a message by appending it to the end of a log
  - a consumer receives messages by reading a log sequentially
  - if the consumer reaches the end of a log, it waits for a notification that a new message has been appended
    - this is similar to how unix tail -f works
  - the log can be partitioned to scale beyond jsut one node

    - different partitions can be hosted on different machines
        * each partition is a separate log that can be read and written independently fro the other partitions
    - a topic is then defined as a group of partitions that carry messages of the same type
    - this is the basis for Apache Kafa, Amazon Kinesis Stream

Change data capture (CDC)

- when a db write happens, the CDC is a stream that takes the event correponding to the db write and sends it to other data strucutres that need to be notified -- eg a search index or a cache -- these are called derived data systems
- db receives write - (usually) commits write - then publishes message to CDC - takes changes to derived data systems
- Kafka Connect is an effort to integrate CDC tools for a wide range of dbs with Kafka