OTHER FORMS OF INTERACTION IN DISTRIBUTED SYSTEMS
(Draft Notes)

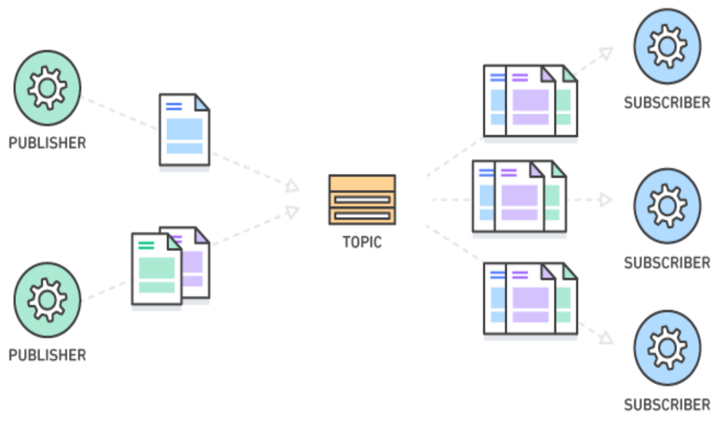
I. Consumer-Producer Communication Model, Pub-Sub Systems, and Streaming Systems

We've focused thus far mostly on one form of interaction in this class: **client-server**. We've talked about basic communication primitives that support this form of interaction (e.g., RPC) and protocols and architectures, and systems built on that form of interaction (e.g., consensus protocols, database systems, service-oriented architectures). The client-server interaction model is suitable for the traditional Web types of interaction: send a request to a server, the server does some processing and compiles a response, and sends it back to you. Then the browser sends another message to check for updates (usually based on a TTL for the page). That's how the traditional "Web" (and traditional applications) have worked for a long time: on-demand services.

But that's changing: the web (and many of our applications) are changing to be more dynamic, real-time, and event-driven. Data changes all the time – breaking news, location-driven notifications, etc. – and the need for real-time notifications in our applications is becoming pervasive. These kinds of real-time, event-driven applications are not well supported by the client-server interaction model.

Another form of interaction in distributed systems is **producer-consumer**. One or more "producers" produce data/events and one or more "consumers" receive them and act upon them. A popular communication primitive that implements producer-consumer is the **message/event queue**. Messages are placed on a queue and available for a single subscriber due to the fact that once a message is read, it's gone. That's not to say the subscriber can't scale out the same process against a queue, but it can't have multiple subscribers see each message on the queue.

Of course, all the systems we've studied, including the one you built, incorporate queues. But they were thus far internal within each node, multi-threading communication, not reliable primitives of communication with scalability and specific semantics. Well, in a producer-consumer-based system, the "queue" becomes the primary way in which components interact.



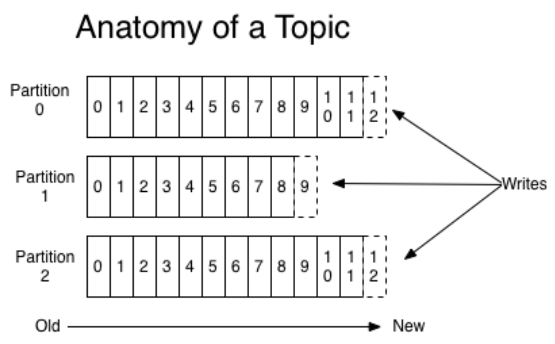
A more general, higher-level coordination abstraction for building real-time, event-driven DS are **pub-sub systems** (publisher stands for producer, subscriber stands for consumer). See figure to the right. Google's PubSub service, Java JMS, Kafka (http://notes.stephenholiday.com/Kafka.pdf), and our own Synapse system (https://roxanageambasu.github.io/publications/eurosys2015synapse.pdf) are examples of such systems. These systems are built on message queues, but they provide much more flexibility and higher level of abstractions (a pub-sub system is for queues what the low-semantic storage systems we studied are for RPC). Pub-sub systems allow for multiple subscribers, sometimes organized in groups, to register for specific types of events. Some systems, including Kafka, can provide guarantees regarding delivery (e.g., at least-once, exactly-once) of these events to the relevant subscribers. Some systems (e.g., Kafka and our own system, Synapse) provide guarantees on the ordering of the delivery of these events to subscribers (e.g., Synapse guarantees causally consistent delivery order). Some systems support sophisticated filtering on "topics" (e.g., a consumer wants to register for events whose names match a REGEX or some other predicate on event properties), but those usually don't scale well.

At an even higher level of abstraction than pub-sub systems in this producer-consumer interaction space is the notion of **a streaming system** (or **streaming engines**). Pub-sub systems only provide the event delivery infrastructure – so a low-level API (akin to how a block-level storage system offers a low level API). Streaming systems build on pub-sub systems but provide higher-level APIs that include support for the computations that process these events (akin to how a database gives support for queries and transactions, which are computations on the data). For example, a streaming engine with relational API supports continuous SQL-like queries for computing, e.g., running averages or counts, etc. There are also streaming engines for online machine learning training. And so on.

II. Example System: Apache Kafka

A popular framework for event-driven programming is Apache Kafka. At its core, Kafka is a pub/sub system built on top of logs, with many desirable properties, such as horizontal scalability and fault tolerance. Since its origins, Kafka has evolved from a simple pub/sub system to a full-fledged streaming platform (see Kafka Streams).

Kafka addresses some limitations of many pub/sub systems as well as some of streaming systems. A characteristic of many pub/sub systems is that they adopt a "push-based" architecture: subscribers register for their interests, and publisher(s) are responsible for pushing the appropriate events to them. This is natural and in line with the interaction model, but push-based implementations often break down when scaling out due to the amount of responsibility on the publisher to handle subscribers and send every message to every subscriber. This also poses problems when subscribers can't accept messages because they are down or falling behind due to load. Another limitation of many pub/sub systems (and many streaming engines) is that individual consumers can only consume each message once. Consumer can usually not revisit previous events. This semantic is not easy to change, because if you want to keep state in the producers, you need reliability.



Kafka addresses these limitations with a **"pull-based" architecture** (consumers check in periodically with Kafka for updates), and with **reliable storage of sharded message logs** (one log per topic) for configurable amounts of time. In Kafka, consumers can scale out to process in parallel via consumer groups (a collection of consumer processes joined by a common group identifier). Multiple consumer groups can consume the same data without having to duplicate it or worry about data being removed after a read. Kafka provides the scalability of queues as well as the multi-subscriber features of a pub/sub system. Kafka also provides better ordering guarantees without having to use the notion of an "exclusive consumer." Lastly, consumers poll for data so if they're down it's not a big deal. As soon as they come back up, they can continue consuming data at whatever pace they choose.

Kafka also offers flexible guarantees on delivery: it supports a variety of delivery modes (e.g., at least once, exactly once). It also provides some guarantees on ordering, though they are at the level of individual partitions of each topic. Please refer to the Kafka documentation for the latest semantics.

III. Microservice Architectures

The producer/consumer form of asynchronous service-to-service interaction enables a different type of system architecture than what we primarily studied in this class. It enables fine-grained event-driven architectures, also named as **microservice architectures**, which decouple applications and services into small, modular components to increase performance, reliability, scalability, and maintainability. A hot computing model these days is the **serverless computing** model, which often uses these kinds of pub/sub systems to construct big, complex services out of loosely coupled and cleanly separated microservices. I encourage you to seek out information about these emerging computing models.

Since this is the last lecture, I wish you good luck in your future career and remember: DS learning is just beginning for you now that you can understand the basics!

So keep on learning!

Acknowledgements

Philip Su's 2018 blog post on Twitter's migration to Kafka:
https://blog.twitter.com/engineering/en_us/topics/insights/2018/twitters-kafka-adoption-story.html

Matt Schroeder's 2019 blog post on why Kafka is popular and how it compares to other systems
https://objectpartners.com/2019/03/13/why-is-everyone-talking-about-kafka/

The Pub/Sub system figure is taken from AWS Pub/Sub documentation: https://aws.amazon.com/pub-sub-messaging/.