REST, RPC, and Brokered Messaging

This article explores three popular styles of communication in service-oriented architectures and how to chose the appropriate style for a given use case.

Considerations

The web is moving towards microservices. If you have not yet adopted this architectural pattern, you are behind. Effective communication between distributed and heterogeneous components is essential for this architecture to work well for your software organization. Monoliths broken down into smaller domain-specific applications afford looser coupling of components along network boundaries. Once those components are defined, the following step is to define how those components communicate across the network. A well-defined microservice architecture should be accompanied by a well-defined communications semantics. Start at TCP/IP and work up the stack. Next, adopt a foundational set of design principles to follow when architecting your system (the Kubernetes community has a good list). Now here are some considerations for how you want your applications to talk:

- Will your application talk to a web browser or a mobile client? Is your application a front-end service?
- Will you expose private or public APIs? Applications deep in the back-end will typically have a private API for internal use only. Find the right security framework to accompany these APIs.
- Will your application need something that HTTP cannot do? HTTP/1.1 is presently the *lingua franca* of the World Wide Web, and HTTP/2 is gaining adoption. If the semantics of neither satisfies your domain language, find a different application layer protocol.

- Does your application need persistent or long-lived connections with other applications for it to do its job?
- How should data flow from application to application? Can data flow one-way or does it need to be bi-directional? Does a traditional client-server model fit (one-to-one)? Is point-to-point request-response sufficient? Will data fanout to several apps (one-to-many)? Will data need to be aggregated (many-to-one)?
- How big is an atomic unit of data exchanged over the network? At what rate is it produced/consumed? This will help drive latency, throughput, and volume requirements for each application.
- Determine a set of service-level objectives (SLOs) even if you have not yet drafted any agreements with your customers. This is necessary to determine how well your system is performing and can be revised when SLAs become formalized.

This is not an exhaustive list (I won't go into things like logging or instrumentation), but it covers some basic ground before choosing the appropriate communication style for implementing your applications' business logic. For weighing other considerations not mentioned above, see Chris Richardson's pattern language for microservice architectures.

Representational State Transfer

Many developers equate canonical JSON+HTTP/1.1 with REST. This is inaccurate. REST itself does not define a communication model but rather a description for how web services can be architected for distributing hypermedia in a scalable fashion. Any application communication architecture that employs the elements and satisfies the constraints of Representational State Transfer is RESTful; but for the sake of simplicity we'll narrow ourselves to the canonical use case where layered data representations are created, read, updated, and deleted (CRUD) via payload exchanges between stateless client-server connectors using URIs and HTTP. By historical accounts, the URI and HTTP standards are direct applications of REST.

REST insists on uniform interfaces. This is fundamental to the notion of RESTful architectures. HTTP has become the *de facto* standard for creating uniform REST APIs. What results is a communication style that inherits the semantics of HTTP. Hypermedia—layered as text, images, audio, video, graphics—are identified, retrieved, and manipulated using CRUD via the operations POST, GET, PUT, and DELETE (we can think of PATCH as a special case of PUT).

Consider Amazon's Simple Storage Service (AWS S3), a paragon implementation of REST . If I want to view objects in my bucket, my client needs to make a HTTP GET operation. If I want to create an object, thereby adding it to my bucket, my client issues a HTTP POST to virtually place that object in the bucket. I can virtually manipulate that object by making a HTTP PUT request. And finally if I want to remove an item from the bucket, my web client sends a HTTP DELETE on the resource referring to the object. What follows is a very clear-cut CRUD semantics:

HTTP request	Method name
POST my-bucket.s3.amazonaws.com	CreateObject
GET my-bucket.s3.amazonaws.com	ReadObject
PUT my-bucket.s3.amazonaws.com	UpdateObject
DELETE my-bucket.s3.amazonaws.com	DeleteObject

A few components are needed to facilitate these interactions. A user agent makes a request to an origin server with the expectation that the server will provide a response. A proxy or gateway mediates secure client-server connections across public and private networks. In the AWS S3 example, a user agent will make a HTTP POST request to the origin server which in turn responds with some indication that the request either succeeded or failed. The request to the origin server may result in cascading interactions with other servers. In the age of monoliths, this might be a single enterprise web server that makes a call to a database server. But in the age of microservices, there could be many servers at play interacting over very complex networks.

Without loss of generality, the CRUD operations are RPC calls whose method names
are: *CreateObject*, *ReadObject*, *UpdateObject*, *DeleteObject*. The request-response interaction when REST is applied to HTTP is itself a kind of remote procedure call whereby the semantics are constrained to CRUD operations. This is a powerful abstraction for dealing with hypermedia over the web. And so REST makes sense for delivering hypermedia to humans and therefore operates closer to front-facing applications: web browsers and similar interactive user agents. This may be the foremost criterion for selecting REST as an architecture for communicating with microservices.

Other criteria are important as well. REST not only makes sense for building front-end web services, it has also become a powerful abstraction for implementing public APIs. Every major successful internet company offers a RESTful API of some sort for fetching and manipulating data resources and web products it owns. Security standards like OAuth and TLS have complemented the progression of HTTP with regard to this success. Because HTTP follows a client-server model, REST also makes sense for one-to-one communication: an individual client makes a request to an individual server that issues a response, granted TCP/IP hides much of the multi-hop mechanisms at the link layer. HTTP has traditionally relied upon closing connections whenever a request-response cycle completes, however with the introduction of server-sent events we are seeing streaming HTTP or streaming REST as a model where connections are kept open so that payloads can be pushed from server to client for asynchronous one-way data flow. Streaming REST can be thought of as a kind of one-to-one pub-sub model. We’ll return to the subject of streaming again in the discussion on brokered messaging.

Remote Procedure Call

Unlike REST in the definitive sense, the RPC style of communication allows for a more specialized semantics but is also less opinionated about agreeing to a standard protocol of information exchange. Rather, clients and servers are stubbed so that remote procedure calls mimic local procedure calls but over a network boundary. And since RPC is protocol agnostic, developers are freer to choose a protocol that aligns with the application domain and its semantics.

Once the focus of software development moves beyond the first tier of services that sit close to the front-end, REST tends to make less sense deeper in the back-end and limits the interactions with the application to CRUD over HTTP. This makes it difficult to build flexible and robust application architectures in scenarios where HTTP is not the prescribed application layer protocol. RPC leaves it open to the application owners to decide which interface defining protocol makes the most sense for the task at hand. As mentioned earlier, CRUD over HTTP is a degenerate kind of RPC—the semantics of which places an artificial limitation on the communication architecture of the system.

The best RPC implementations aim to be protocol agnostic. If HTTP makes sense, then use it. But what about applications that do not support HTTP? Consider the scenarios of writing to a cache using Redis serialization protocol

```
// Redis protocol as RPC interface for caches
// We can think of Set() and Get() as RPC calls
err := cache.Set(key,value).Err()
val, err := cache.Get(key).Result()
```

or of fetching table rows from a relational database over JDBC.

```
// JDBC protocol as RPC interface for databases
// We can think of executeQuery() as a RPC call
Statement stmt = dbConn.createStatement();
ResultSet rs = stmt.executeQuery(sql);
```

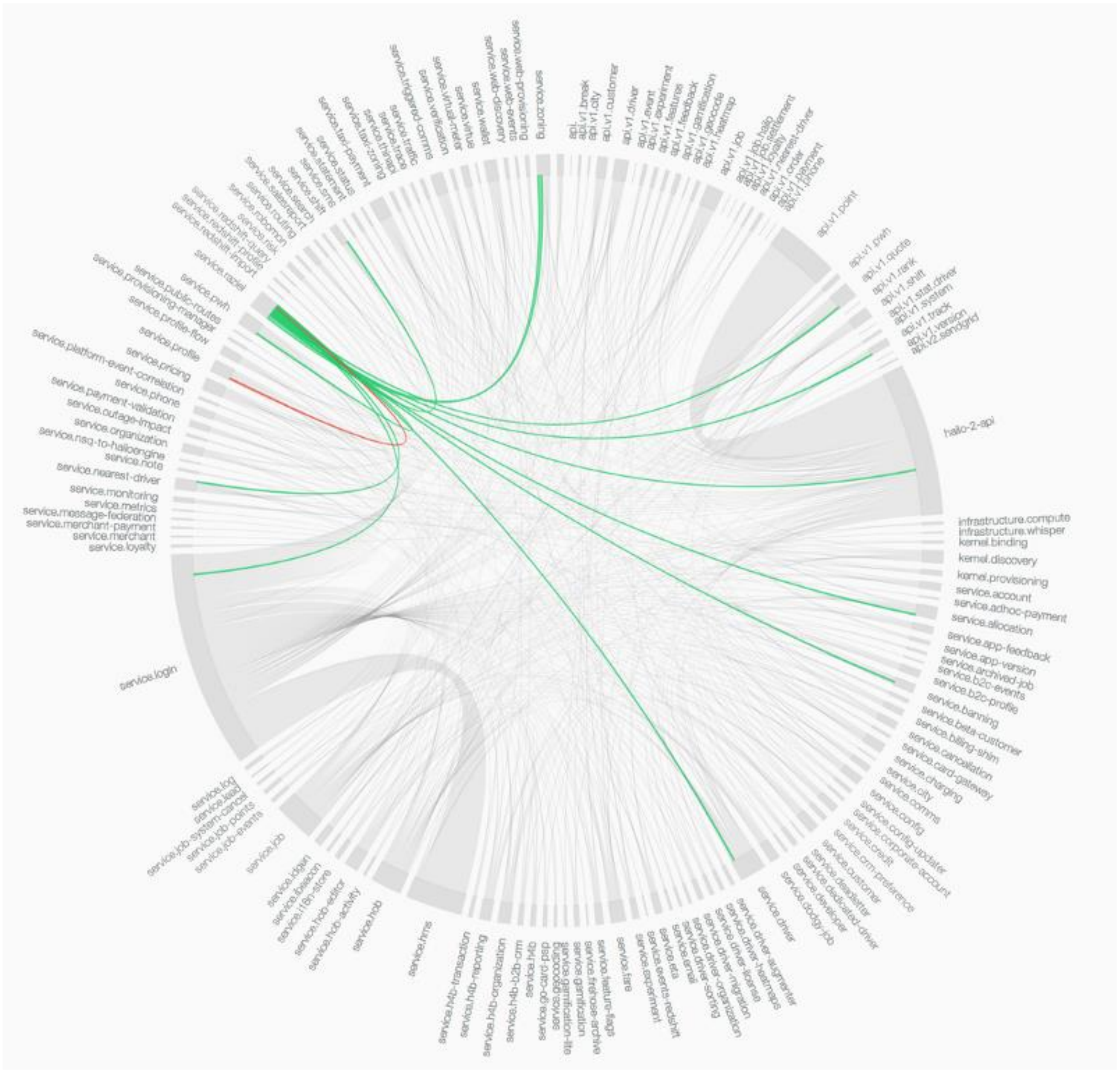
The authors of each of these applications implemented their own domain-specific but standard communication protocols. And although these communication styles are not RESTful, they fit the RPC model.

There have been attempts to unify these implementations under a single RPC system. Twitter's Finagle libraries are exemplars in this regard, and gRPC by Google and Square represents an incremental step in the progress of scaling RPC for the cloud. Microservice architectures embrace protocol agnosticism in the same way they embrace polyglot engineering. The end goal is to liberate both the developer and the application all while delivering software fast and deploying scalable, reliable distributed systems. Thus, RPC is a part of this embrace, especially for applications like caches, databases, and intermediary services which tend to be deeper in the stack.

I'd argue that inasmuch as REST is the most appropriate communication style for distributing traditional hypermedia content over the Internet (namely between servers and web clients), RPC is the most appropriate communication style for microservice architectures within the data center. In its simplest incarnation, RPC is a sequence of (1) issue a request, (2) wait for a response, and (3) process the response, but unlike REST it allows for other application protocols to be used. RPC also relaxes the client-server constraint. A server can be a client, and a client can be a server. Services might be chained together:

```
Service A  <=>  Service B  <=>  Service C  <=>  Service D
```

And so ***Service B*** not only serves ***A*** its requests but is also a client of ***C***, and ***Service C*** not only serves ***B*** but is also a client of ***D***. Relaxing the client-server constraint provides an equally immense degree of freedom as protocol agnosticism because RPC can span arbitrarily many applications and employ arbitrarily many protocols. The request-response nature of RPC, in addition to its simplicity and maturity, means that it can be traced well in large-scale distributed systems with readily available instrumentation tools, as formalized by Google's Dapper and demonstrated in open source projects like Zipkin and efforts like the OpenTracing spec.



Observing RPC calls traced in a large distributed system (https://sudo.hailoapp.com/services/2015/03/09/journey-into-a-microservice-world-part-3/)

The freedom of creating your own semantics is also important and is essential to domain-driven design. It’s the old hammer-nail pitfall metaphor. REST, exemplified by CRUD over HTTP, is just one tool; but it shouldn’t be applied to every domain. Let’s look at some examples. Consider container orchestration. Applications like Swarm, Marathon, and Kubernetes use RPC to build a semantics around container scheduling: *CreateContainer*, *StartContainer*, *StopContainer*, etc. Consider warehouse automation. I may want to build some robotic systems that respond to actuation triggers like *PackageItem*, *PlaceOnConveyer*, *ShipProduct*, etc. Or consider the Internet of Things (IoT) space. At Tesla we control IP-connected battery packs, so I may want to dispatch RPCs like *ChargeBattery*, *DischargeBattery*, etc. Each of the methods may have parameters which are all passed as input when issuing the RPC. The key take-away is that my domain language for defining interactions with services—across network and process boundaries—can be made richer with RPC and is not limited to the semantics of *Create*, *Read*, *Update*, and *Delete*.

RPC is still very much a one-to-one communication model. Although a RPC server stub may be implemented in a way where the RPC runtime invokes a single method but issues fan-out requests to multiple server instances, the implementer is left with the burden of discovering, identifying, and naming each one of those instances. Transparent proxies like Linkerd and Envoyhandle this burden to a limited extent, and it is common to employ RPC for communicating with cluster instances in this way. Applications like Cassandra, Spark, and Kafka have some kind of RPC layer for coordinating nodes across the cluster network. These RPC mechanisms are generally one-to-one or one-to-many. And so fan-out can be achieved using RPC albeit in these highly specialized applications. Fan-in most certainly cannot be done well without making some painful design choices. The data stream abstraction comes to mind in generic fan-in / fan-out graphs. With the introduction of HTTP/2, developers can build streaming RPC applications with bi-directional data flow; but this also is limited to point-to-point communication.

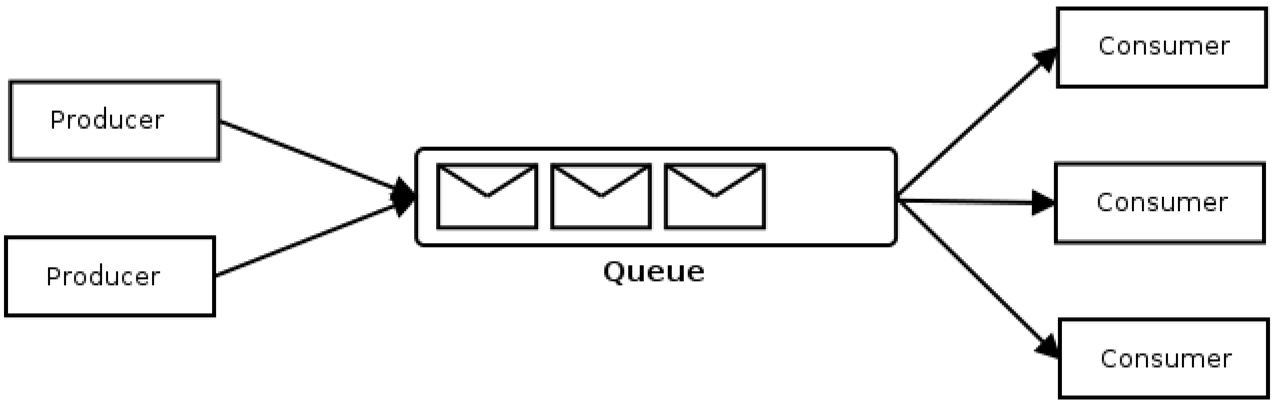
Brokered Messaging

REST and RPC do not work well for streaming large volumes of data at high throughput rates in pub-sub communication systems where it is desired to abstract away the locations of producers and consumers. In the pub-sub model, producers publish to one or many topics and consumers receive on one or many topics. Producers and consumers only need to know the name of the service brokering the connections and a list of desired topics. Messages are queued by the service broker. The message broker mediates one-to-one, one-to-many, and many-to-one interactions. And thus extremely large and highly performant push-based fan-in / fan-out systems can be built based on the Message Broker pattern. This becomes heavily useful for asynchronous communication, unreliable networks, and big data applications.

Communication is asynchronous in message-oriented middleware. A key characteristic of asynchronous messaging is that producers and consumers make persistent connections to the message broker. This is much different from REST and RPC (except for streaming) in which communication is synchronous: a request followed by a response before closing the connection. Message brokering offers an advantage over REST and RPC in circumstances where the cost of maintaining a persistent connection for asynchronous communication is less than the cost of making several connections for synchronous communication. This makes message brokering a popular choice for chatty web services in which data are produced and consumed at a high frequency. This is not much different from streaming REST or RPC, but another key distinguishing characteristic is that producers and consumers are further decoupled by placing a broker between them. Sources and sinks of asynchronous data flow are completely abstracted by the message brokers. The system architect must make tradeoffs between introducing the cost/benefits of asynchronous messaging, the cost/benefits of persistent connections, and the cost/benefits of decoupling producers and consumers.

Asynchronous messaging enables efficient communication between web services and devices over unreliable network connections. Message brokers form an essential backbone for the IoT where devices and network connections can fail and repair themselves randomly. In IoT applications, services need to communicate with devices even when a device is unreachable. Since messages are queued in a broker, reliable communication can be accomplished by making brokers push message off the queue when a connection is established or by making consumers responsible for pulling messages off the queue. Furthermore, the broker can facilitate the exchange of acknowledgements between services and end devices.

Moreover, asynchronous messaging decouples requests from responses. Requests and responses can be thought of as message types. A benefit of the message broker pattern over REST and RPC is that "request" messages and "response" messages can be queued, and the queue can be shared with several producers and several consumers. In REST and RPC, a session can timeout if a response is not heard from the requestee within some configurable period of time. Communicating with devices using a REST or RPC medium over the web would be inefficient because the cost of retrying a session and of repeated re-connections is higher than the cost of resending a message with a persistent connection, especially if communication is very chatty. Also in order to use REST or RPC within IoT, the device would need to implement a server stub. This would immediately break the REST architecture since the device is also a client. And although the client-server constraint is relaxed in RPC, the IoT landscape involves several one-to-many and many-to-one communication interactions, which makes RPC an awkward (but doable) fit. Device control and monitoring in IoT applications benefit from message brokering due to these reasons.



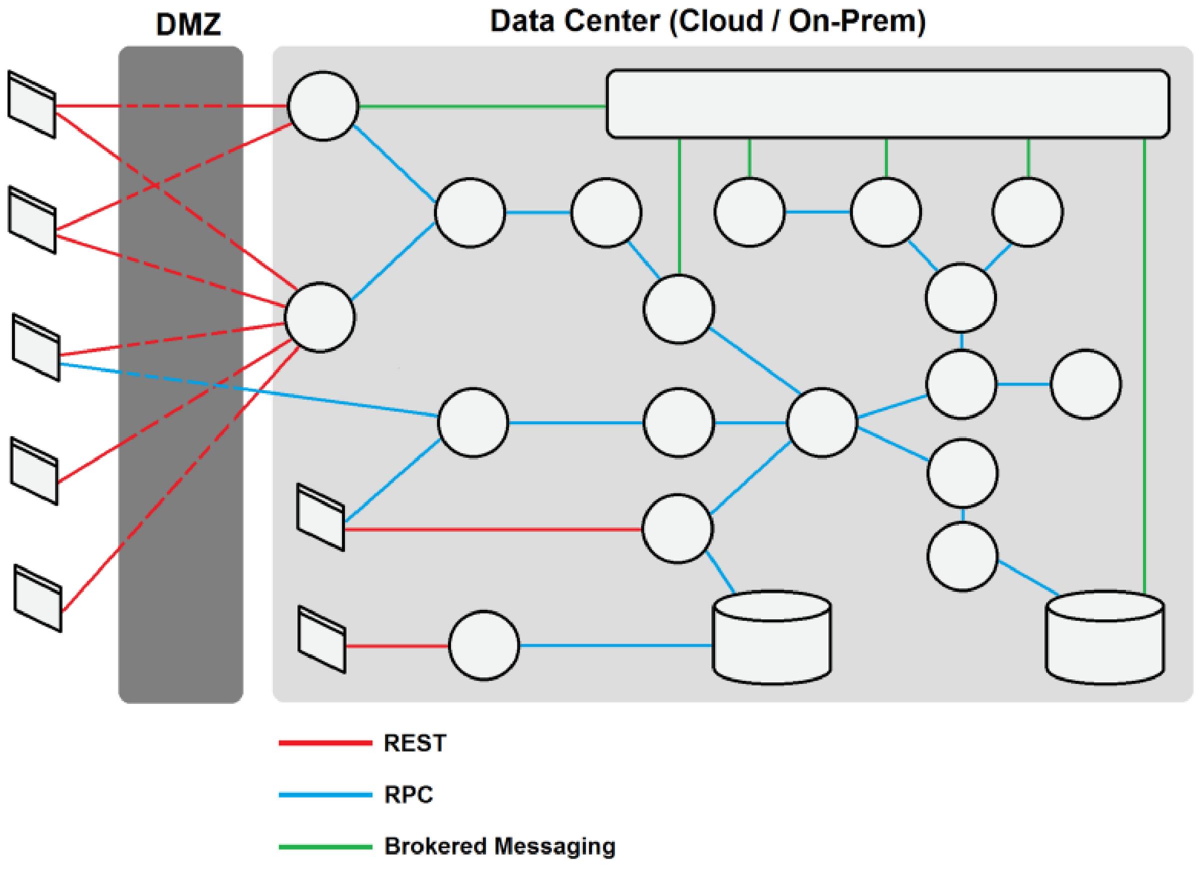
The basic Message Broker pattern (source: http://activemq.apache.org/clustering.html)

The rise of big data applications has brought renewed importance for message brokers to the enterprise. Traditional enterprise messaging standards like the Java Message Service (JMS) and Advanced Message Queue Protocol (AMQP), which appeared in the late 1990s and 2000s, were not intended for big data. Today large scale data processing is achieved using message brokers like Kafka or services like Google Pub/Sub and Amazon Kinesis wherein the assumptions about data warrant greater consideration for volume and managing that volume. Today, high throughput data aggregation systems can be built using message queues instead of writing scripts that fetch data over point-to-point protocols. The same is true for performing large ETLs in modern data warehousing systems like Hadoop. Spark and Flink can be coupled with message brokers for large-scale, low-latency distributed stream processing. Many of today's click-stream analytics and recommendation engines are built on top of these. Online machine learning systems rely on message brokers for updating classification and regression models in near real-time. Message streaming enables the construction of high-throughput data pipelines assembled in directed acyclic graphs. The message broker is at the heart of architectural patterns like CQRS and Event Sourcing. And algorithmic, high-frequency trading platforms have both pioneered and borrowed from advances in high-performance message queuing systems.

The message broker is its own organism in the world of microservices. REST falls short of the requirements for moving large data in streaming applications. It is an outdated architecture with respect to these developments, but it is well-understood and well-adopted. And companies like Twitter, Square, and Google have established vetted RPC practices for microservices. However, it is yet to be seen how best practices emerge for message brokers against this backdrop. Where do kappa and lambda architectures meet the microservice architecture? The internet giants have no doubt answered this for themselves using their own proprietary ecosystem of libraries and tools. Open source utilities for message tracing, visibility, instrumentation, and reliability are lacking or are foisted upon application developers to implement on their own. The next phase in the evolution of high performance message brokers will be tailoring extant design patterns and practices to blend with the microservice paradigm.

Conclusion

REST, RPC, and Brokered Messaging are not mutually exclusive; they can all work together in your microservice architecture. Every successful cloud-based tech company employs these communication styles effectively to some degree. We have covered the different cases and circumstances in which each style comes into play and where each are appropriate. What remains to be seen in the upcoming technology iteration cycles will be how the different communication styles resolve along adjacent domain boundaries in modern service-oriented architectures. For example, how will a service that exposes a REST API consume data from another service that exposes a RPC API or a messaging API? What semantics need to be preserved along these boundaries? Again, this is something the internet giants have already decided within their own businesses, but those outside the high castles and the rest of the developer community at large are left to figure this out for themselves.



REST, RPC, and Message Brokering working together in a distributed system. In this diagram REST services most browser-based UI interactions, particularly over public networks. RPC makes up the majority of inter-service communication as well as a few UI requests. And messages are brokered across services and data stores that rely on pubsub streaming. Each component satisfies a domain and communicates with other components using a domain-specific semantics.

Software relentlessly evolves. Best practices for service-oriented architectures today may quickly become outdated tomorrow. Developers and system architects working in environments undergoing constant flux should care the most about patterns that anticipate and endure the cycles of technological change. Good foundations must be solid enough to withstand these tests of time.