Thread Architecture

immediate

name	Stage with either fixed number of threads, or has a limit on the maxim number of threads running.
name	Stage without limits on number of threads. A new thread is spawned whenever needed.
	Scheduling point, where requests are queued.
	Asynchronous relationship between stages: upstream stage returns as soon as it hands off request to the downstream stage.
	Synchronous relationship between stages: upstream stage blocks until the downstream stage completes the request.
	Bypass path, where request bypass the regular processing path and scheduling point.
	Machine boundary. Communication between different machines consumes network resources.
	Different types of resources. Legend shown in a stage box means that stage consumes the corresponding resource.

Table 1: **Legend Summary.** *Legends used in thread architecture graphs.*

1 Thread Architecture of Popular Systems

In this section we show the thread architecture of some popular systems, and point out the scheduling problem derived from their architecture. Table 1 summarizes the legends used in the architecture graphs. Note that we view each queue as a (potential) scheduling point. Even though scheduling may not be explicitly implemented, taking requests from a queue itself is a form of scheduling. We omit the stages that are not resource intensive, or not active in normal execution path, due to limited space. We also omit background activities, e.g., compaction, which could be resource-intensive, but can be managed by different mechansims, such as throttling or utilizing system idle time.

1.1 Cassandra

The thread architecture of Cassandra is shown in Figure 1. In Cassandra, all nodes play an identical role in a "ring" architecture, and data are replicated in multiple nodes in the ring. When clients send queries to one of the Cassandra nodes, the following sequence of actions occurs(the number labels in Figure 1 correspond to the numbers in the list below):

1. In the Request Handling stage, threads asynchronously read client's requests, decode them, and process them until completion. After parsing the message to request, a thread first looks up where the relevant data are stored. For local data , read/mutation requests are directly submitted to the corresponding stages locally. For remote data, the coordination thread passes messages that contain requests to the MessageOutgoingService stage. It then blocks until the request completes (i.e., step 6 finishes).

2. The MessageOutgoingService stage, which spawns three threads for each TCP connection (for small, large and gossip message respectively), picks up the messages and sends them through the network. On the receiving end, the MessageIncomingService stage reads the data off the network and de-serialize them. MessageIncomingService spawns one thread for each connection.

3. Once finished, MessageIncomingService puts the parsed message into the queue of different processing stages (Read, Mutation, etc.), based on the message type. These processing stages execute the tasks, which might include looking up the cache, performing I/O, and compressing/de-compressing the data.

4. After the requested actions on data are completed, a response is generated and passed to the MessageOutgoingService stage.

5. The MessageOutgoingService in the remote node now sends to the response back to the coordination node, which is received by the MessageIncomingService stage.

6. MessageIncomingService passes the response to the Request Response stage, who is responsible for executing any callbacks associated with the request completion.

7. Once the request is completed, the coordination thread passes the response to the Response Stage, who is responsible for serialize the response and sent it back to the client.

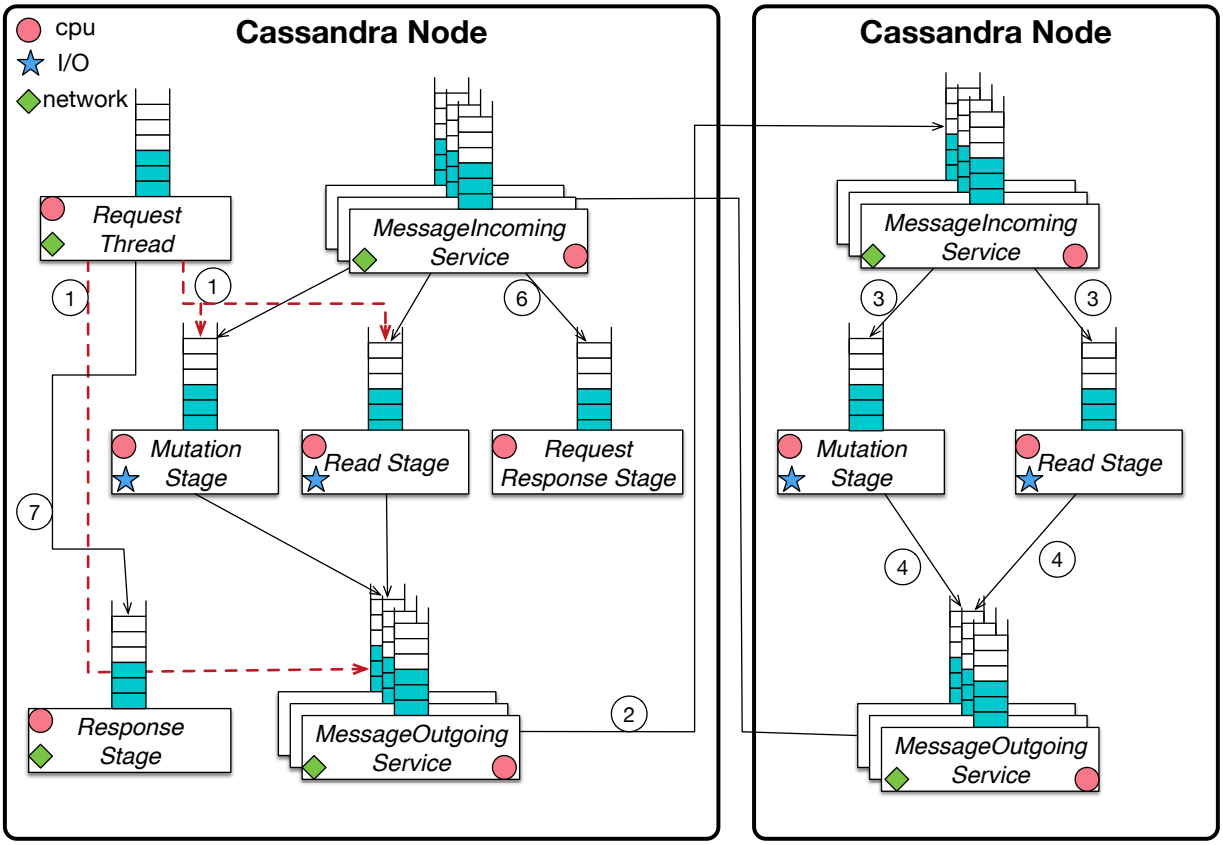

Figure 1: **Cassandra Thread Architecture.**

References

[1] Cassandra Issues: Move away from SEDA to TPC. https://issues.apache.org/jira/browse/CASSANDRA-10989, January 2016.