Radiator Labs' Questions

Sarah Peck

September 17, 2019

- Describe how you would accomplish updating 100+ devices with the same firmware.

  There are several ways to extend my code to accommodate more devices. There is the breadth-first type of approach where for each chunk of firmware, we would send out the chunk to all devices before moving onto the next chunk. This may reduce processing time on the master server, by needing to queue up each message only once before sending it, but would lead to a long time between starting to send the firmware and every device completing its download. This approach is not particularly efficient in terms of the receiving device, and could lead to long periods of the device not receiving or processing any firmware.

  A corresponding depth-first approach would be to complete the entire process for each device before moving to the next one. This reduces the time between the firmware starting and finishing the download process for each device, but could lead to backups and conflicts if done serially. Some devices may not receive the firmware if devices ahead in the queue are slow or problematic.

  My recommendation would be a parallel processing solution where there are several masters sending firmware updates to prevent slowdowns due to a minority of devices, but still using a depth-first approach to minimize the time for each device to download all the firmware. Since that is the part of the process we would not have control over, it is the part where I would want to minimize risk as much as possible. That said, this is the most resource-intensive solution for the master. Depending on resources and the priorities of the situation, some mix of the above options may be appropriate, for example, using a breadth-first approach on ten devices at a time. With more time and sophistication, caching the messages so that they are processed only once per message may also be a way to improve efficiency without sacrificing performance or downtime.

- Imagine the devices respond to messages over a different channel than an HTTP response. For example, imagine the server responds to every valid message to a device with 200, and the devices actual response arrives asynchronously over a websocket. What architecture would you use?

In this case, a serial method of sending each message only after the previous message has been successfully received is unsustainable. To deal with the asynchronicity, each message should have some identifier along with the payload so that when the device's actual response arrives, it would be possible to identify which messages have been successfully transmitted. This system would also need more metadata for the whole transmission to order the messages properly before they are installed. In this case, if there is any kind of synchronous communication at all (the 200 response for example), it may make sense for the server to send each message after receiving the 200, but before receiving the asynchronous validation. Once all the messages have been sent, the websocket could be checked for failures and the server could go on to resend the failed messages. If there is more processing power available, the server could check the websocket earlier, or send all messages before receiving any 200 responses.

- In addition to updating the firmware for 100+ devices, imagine each device takes 30 seconds to respond to each message. Would this change anything?

  This would certainly change the code I built, because it waits for each message chunk to be successfully received before sending the next one. With a long message, this time would add up very quickly. In this situation, the breadth-first approach of the first question may make more sense, because during the 30 second downtime between sending messages, the server could send messages to more devices. In terms of the architecture I described in the second question, I would not check the websocket during the first 30 seconds after sending a message, and I would not wait for 200 responses before sending the next message. If there is enough processing power, it may make sense to send all of the messages before getting any responses, and then "clean up" whichever devices or messages were unsuccessful. Another option would be to send a test message to see which devices are responding, and then send all the messages to just those devices that are functional at that time. As always, much of this comes down to competing priorities and resources.

- Imagine that in addition to performing firmware updates to devices over a REST endpoint, you also need to communicate with devices over other protocols like MQTT, CoAP, or a custom protocol over TCP. How could your design accomodate this?

  My design would remain much the same. There are only a few functions specific to the communication protocol, so incorporating other protocols would be a matter of creating similar functions for them and selecting in the code which protocol function is appropriate. My code could also be further modularized to abstract out specific error messages to make it more extensible. Using inheritance would be a good way to help the code mature by defining properties specific to each protocol individually, like the maximum length of a chunk. For a code base as simple as the

one I designed, the risk would be in making the code less readable and comprehensible for other people to use.