Specifying Asynchronous Device Behaviour

Peter Seymour
peteralanseymour@hotmail.com

February 9, 2020

1 Introduction

In [1] a pattern was discussed for passing messages to an asynchronous device. While it gave some guarantees such as being wait-free and establishing conditions under which send failure was possible nothing was said about how the device should act. This paper aims to demonstrate a small set of parameters that enable efficient behaviour using soft constraints.

2 A Basic Protocol

Once a device has been started it will be subjected to a stream of messages some of which may require a response or a particular duty to be performed. A disk device must store or retrieve data for instance. When synchronous requests are made it's fairly easy to know what the device must do, that is, fulfil all the requirements of the request as fast as possible so that resources are free for other clients to make requests. However, the asynchronous case allows a far greater degree of freedom. A disk device can buffer many requests together and process them as a batch for greater efficiency. Simple elevator algorithms are many times more efficient in the average case for hard-drive access but may produce poorer results for any individual request. This is a price worth paying when processing the data cannot commence until it is all loaded but for streaming audio this would be a worse situation.

As mentioned in [1] there is a timeout for any client sending a message to a device. This timeout serves as an approximate upper bound for the amount of time the device can postpone processing a message. Although the client may not expect a result for a considerably longer period of time if it consistently takes too long then once the message queue is full the client will experience a timeout. The simplest example is the consumption of bursts of messages. If the client is sending messages on average faster than they can be consumed the timeout is critical. The device can postpone processing the first message of any burst so long as it makes up time processing the rest.

This leads to the result that extending the size of the message queue only changes behaviour over a short time frame. Any long term failure to consume

messages fast enough will eventually use up even the longest queue. The queue length therefore serves as a smoothing effect and should be of the same order of magnitude as the length of any burst. The device is configured with a *latency* parameter that specifies the amount of time it can postpone responding to a message. This should be an order of magnitude smaller than the *timeout*. The *latency* parameter allows a device to gather messages and process them in bulk so long as the total time is roughly no greater than this period. An alternative strategy for the device is to become inactive when the message stream dries up. So long as it periodically wakes up at intervals of length proportional to *latency*. This saves resources when the device is idle. Once the deivce has started to process a burst of messages it needs some indication of how fast it should be expected to perform. This is again similar to *latency* but rather than wishing to become inactive after each message it will wish to buffer a fixed number before processing. To specify this a minimum *rate* parameter that indicates how many messages should be consumed per second is given. This enables the device to gather a batch of messages proportional to the *rate* and the *latency*.

3 Example

This examples looks at the simple case of sending characters to a terminal device that runs as a single process. It is expected that the stream will exhibit burst like behaviour as strings of characters are sent and there are periods of idle time when the server does other work. The device maybe running remotely and can poll its end of the message queue as it wishes. Since the expected time from the server sending a character to it appearing on the screen is governed by the *latency* parameter it may go into an idle state only to check after this period of time to see if the queue is now non-empty. Once it discovers it is non-empty then the device can consume characters and write them to the screen as fast as it can. Only when the queue becomes empty again will it go into an idle state.

This behaviour ignores the *rate* parameter as the device works as fast as possible. If it underperforms there is nothing it can do about it and will rely on the *timeout* being sufficiently large to throttle the server as the queue fills up. However, the terminal will also need to do other work such as respond to user input. The user may wish to reconfigure the display so the terminal device must turn its attention to that. It must now decide how much time to devote to both jobs. If the message queue is empty it sets an alarm to interrupt it after a *latency* period and will attend to the user. Once the interrupt triggers and it sees messages in the queue then it must process at least an amount proportional to $latency \cdot rate$. This is all that is expected of it and it can then resume dealing with the user.

If these approximate guarantees were not in place then one of the user and server will be arbitrarily selected as having the highest priority. This is a situation that could lead to starvation and is hard to get right without any sense of expectation. The server can swamp the device with messages to the extent that the user sees a frozen set of controls which fail to respond. In the alternative

case the server will be more likely to experience a timeout when there is rush of user activity even though some of that could harmlessly be delayed.

4 Conclusion

There are certainly no hard guarantees but neither could there be. It would be possible to specify a system of constraints for which there is no solution. Even if a solution could be found using this as a subsystem implies constraints on the larger system. Specifying these constraints would also be complicated by their abstract nature. The two parameters outlined in this paper correspond to desired behaviour of the system which is after all what we are looking for. Problems may still arise since deciding whether two numbers differ by an order of magnitude is subjective. However, with extensive testing there is now a framework in place for evaluating which parameters are important and deciding how they should be changed.

References

[1] "A Pattern For Device Drivers", P. Seymour, 2007.