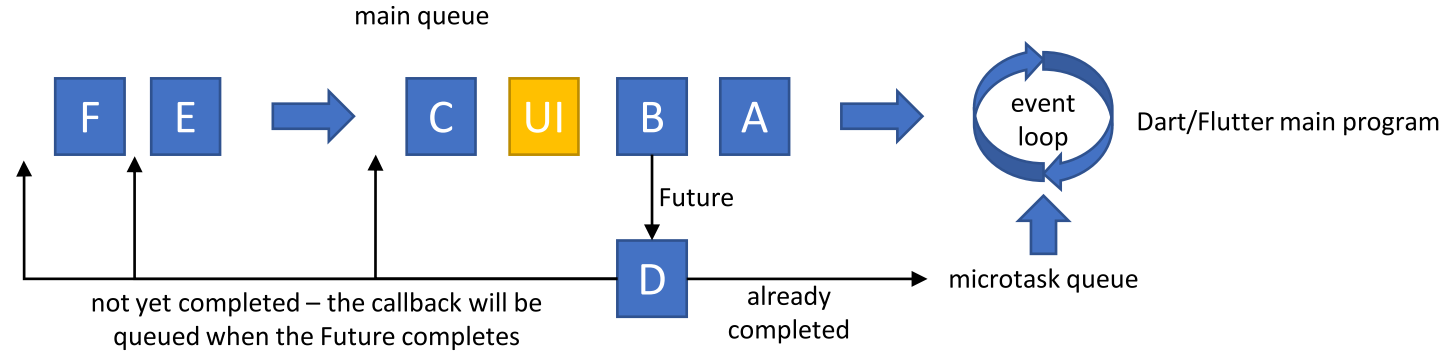


Possible orders of execution

A B UI C D completes before E E F

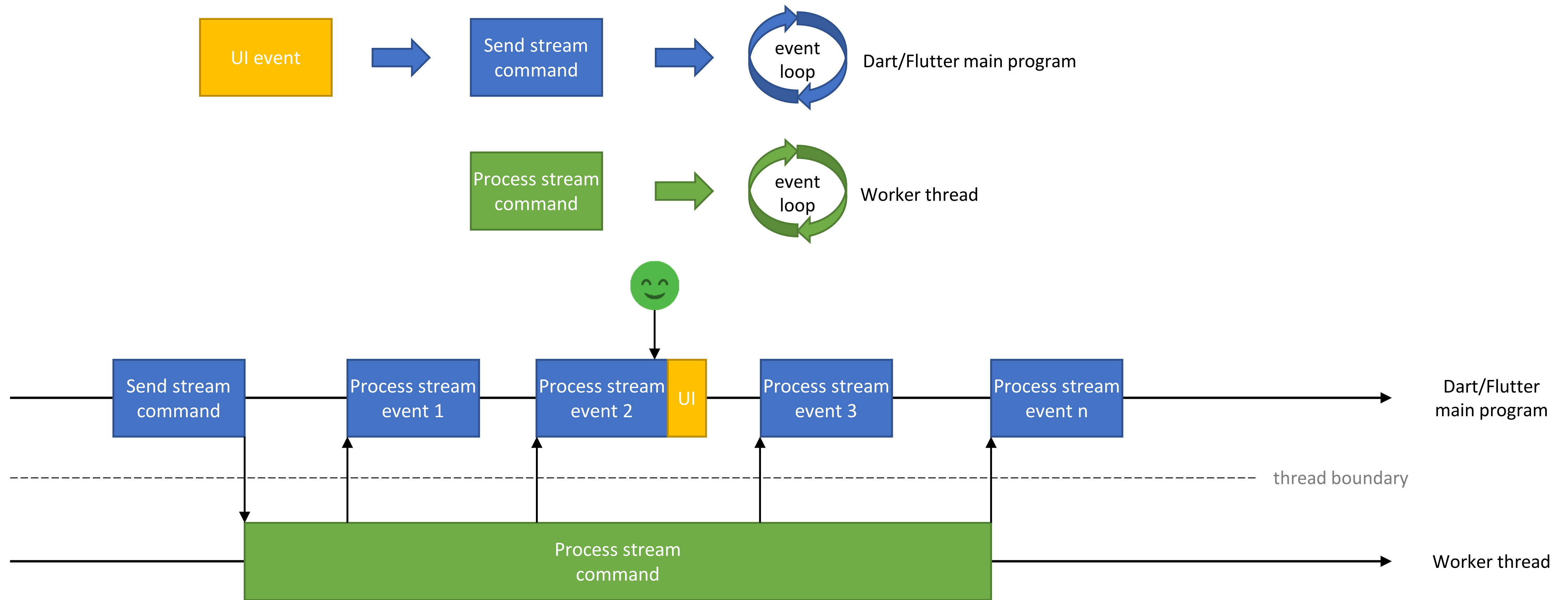
Dart/Flutter main program

A B UI C E D completes after E/before F F

A B UI C E F D completes after F

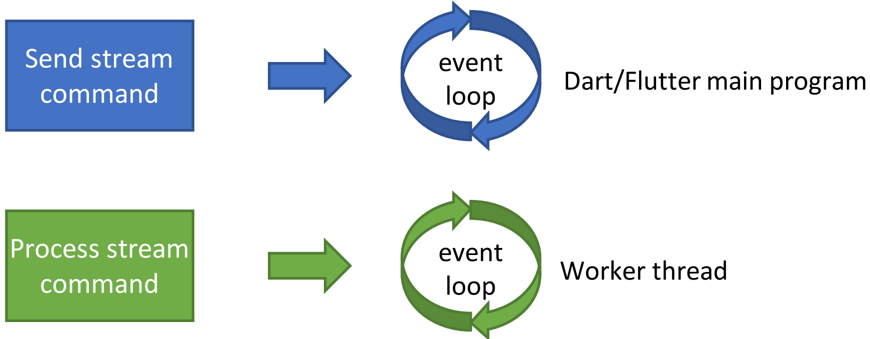
A B D (already completed) UI C E F

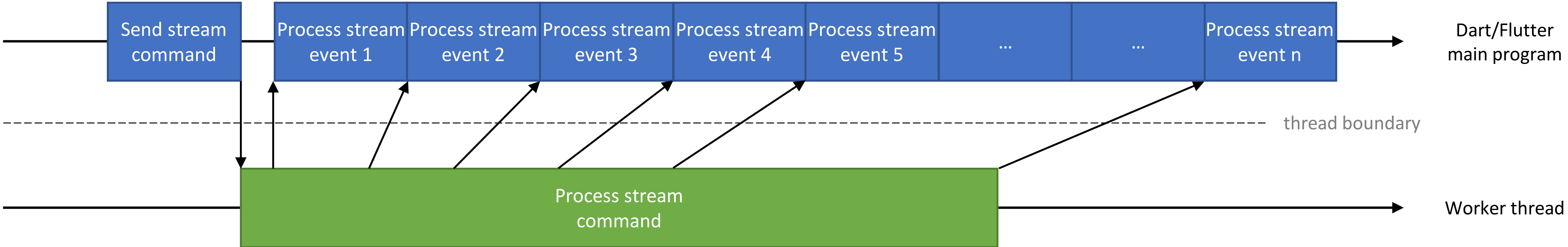
Dart is based on an event loop which processes events from two queues: the “main” queue and the “microtask” queue. Events are queued as they arrive, and the corresponding tasks are processed in line (first-in/first-out). Events from the microtask queue are processed in priority and events from the main queue will only be processed when the microtask queue is empty. Adding events to the microtask queue is usually reserved to Dart’s core runtime, but user-code can call the “scheduleMicrotask” function to explicitly have some of their code run from the microtask queue (to be used with care: overusing the microtask queue will delay processing of events in the main queue, or even prevent them from running at all if additional microtasks are created!). When a Future is created by a task, it is queued after the existing events and the corresponding code will be executed when the Future has completed – with an exception: Dart will queue already-completed futures to the microtask queue.
E.g. if D is waiting on a Web service call, it will be executed when the Web service has responded, only after all other queued events have been processed. If D was created using “Future.delayed(Duration.zero)”, it will be queued after C. If D has already completed, it will be executed on the microtask queue after B.
This explanation might still be simplified with possible corner cases! See https://web.archive.org/web/20170704074724/https://webdev.dartlang.org/articles/performance/event-loop (apparently bugs 9001 & 9002 have been fixed).



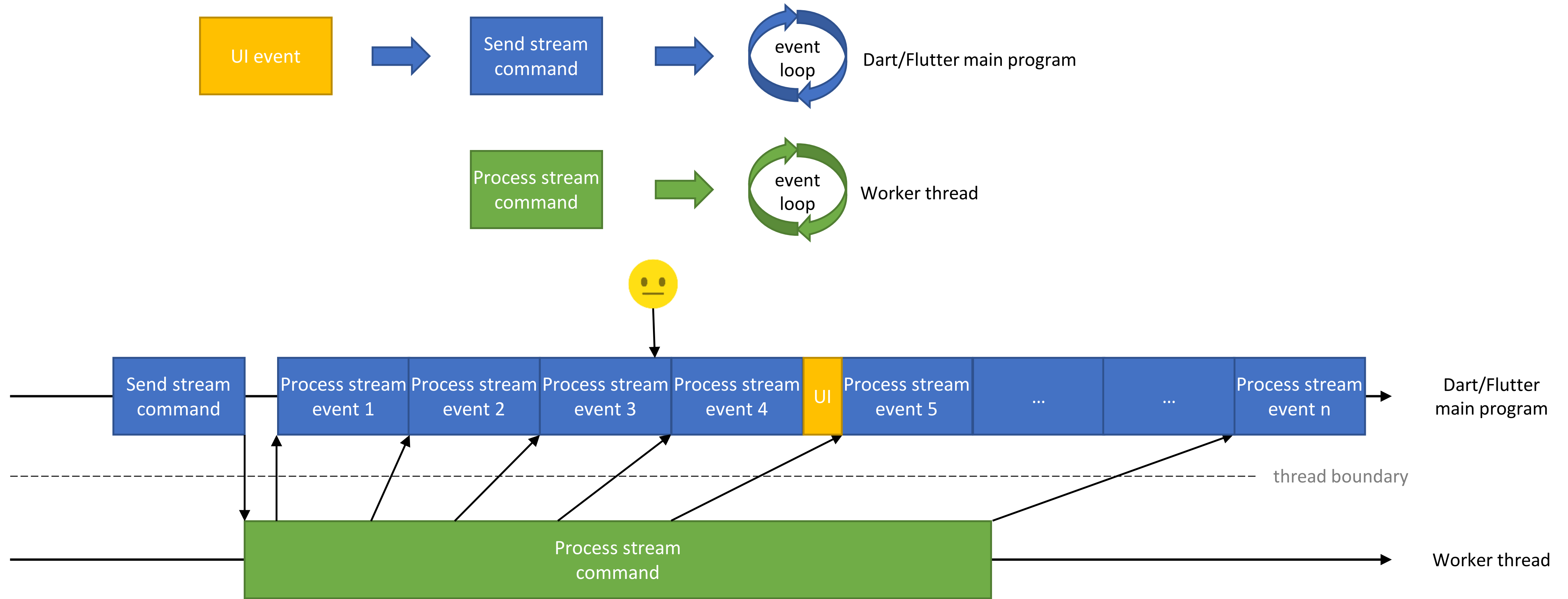
Executing long running tasks in a separate thread will free the event loop of the main program, so it can respond to UI events in a timely manner.
Responsiveness will depend on how long it takes to process stream events.

Crossing thread boundaries does not come for free: data must be cloned.
Depending on the data shape and the platform, cloning can damage performance (Web platform is worse).
Using Squadron/Web also has an impact, as Squadron/Web inspects the data structures too.
This step can be disabled by setting inspectRequest/inspectResponse to false when invoking the worker service.

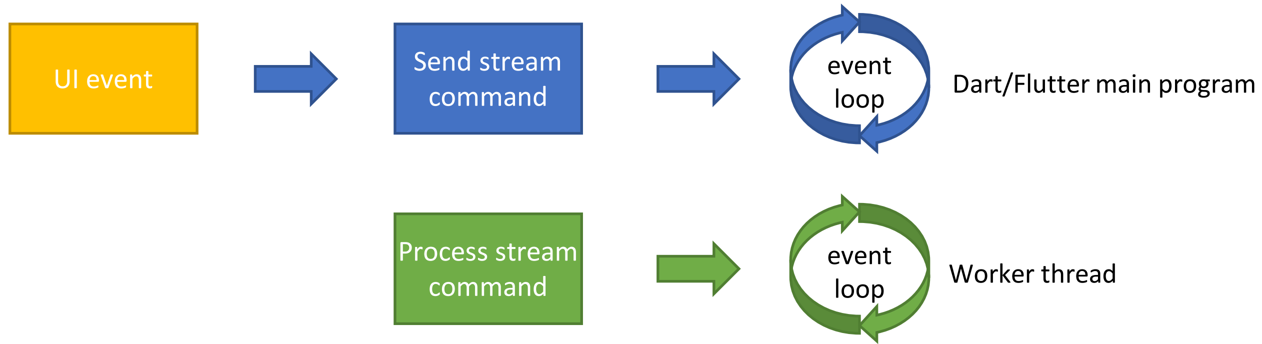




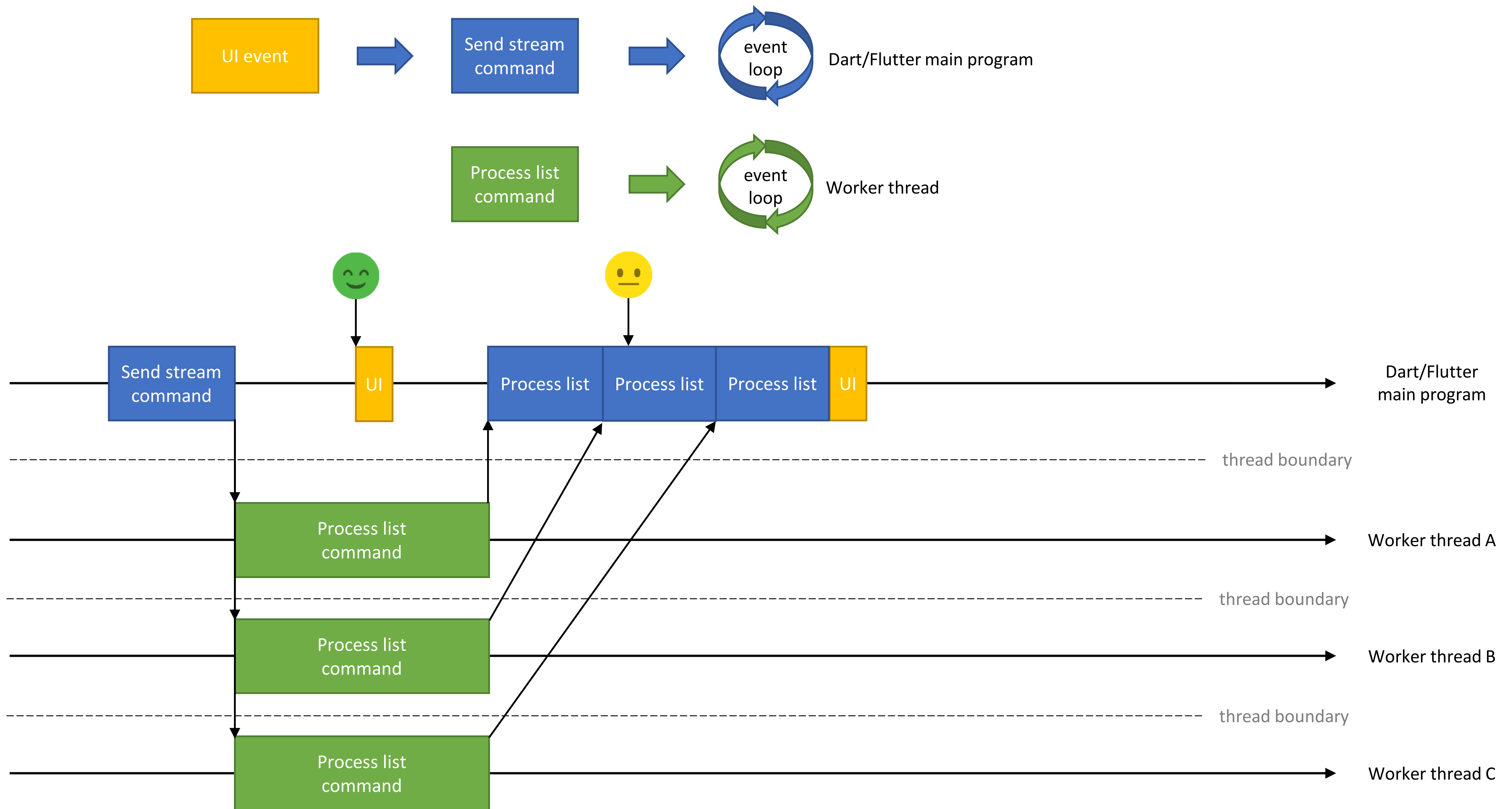
When the stream produces events faster than they can be processed on the main thread, it floods the event loop.



Since the event loop's queue is already filled with stream events, UI events are queued after stream events.
UI events will be processed after the existing stream events, causing UI janks.



Parallelizing makes things worse as the main event queue gets flooded even faster with interwoven stream events!



To balance performance & responsiveness, it's probably best to avoid streaming in this kind of scenario. The problem is because custom objects cannot cross thread boundaries (at least on Web), they *must* be created in the main event loop. At the end of the day, to ensure best performance on VM, platform-specific services might be required to take advantage of Isolates' relaxed constraints on data, while keeping it working with Web Workers. Or the program could stick to using bare Map structures instead of custom objects.