# Introduction

Sequencing is an aspect of program design.

Sequencing describes the flow of control in a program, e.g. answering the question: when a program element executes, what element will be executed next?

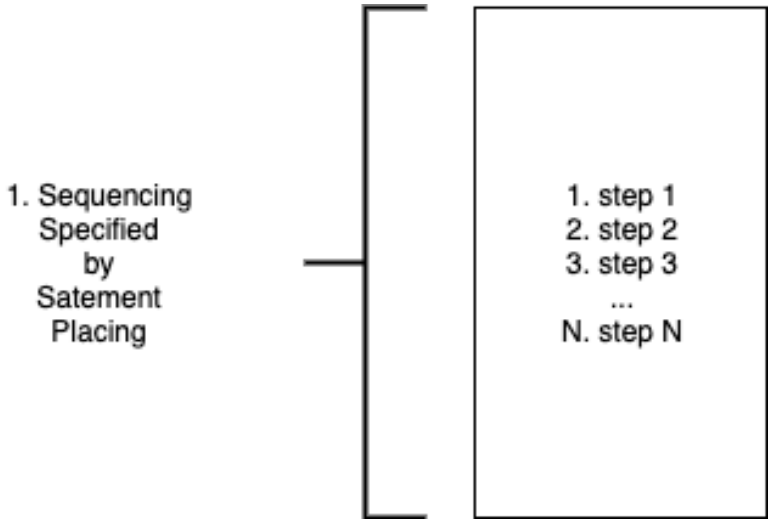
Most languages encourage *statement* based sequencing.

Another common sequencing style is the use of full preemption - threads of code execute in a synchronous (statement based) manner.

The operating system's *dispatcher* decides that a given thread should run.

The use of synchronous sequencing is a design choice. Other choices are possible. I list some of the possible choices and attempt to draw diagrams of their flow.

# Statement



Statement-based sequencing is common in most current text-based PLs[1].

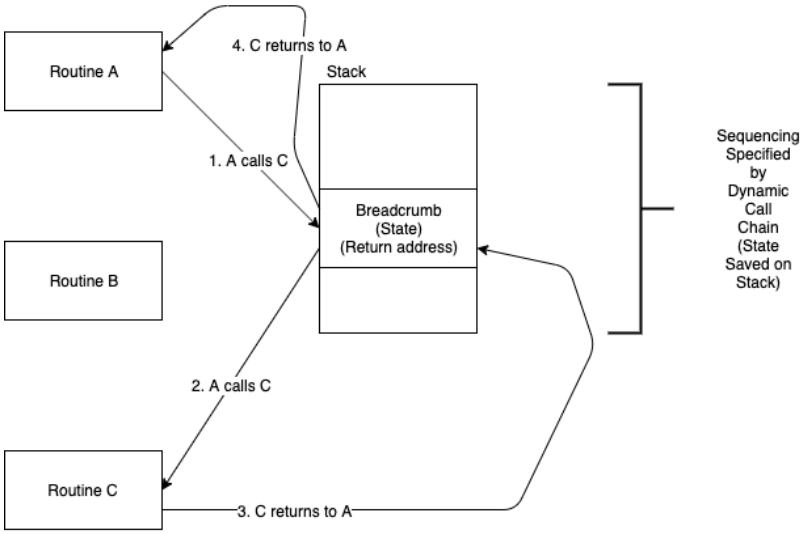
Statements are executed in an order based on their textual appearance.

Subroutines are executed in stack-based sequencing. (See "Stack").

Statement based sequencing is caused by text-only syntax.[2]

Routines are called in a synchronous manner.[3] See *stack*-based sequencing.

# Stack




Stack-based Sequencing

Stack-based sequencing is the common form of CALL/RETURN.

[1] Programming Languages
[2] A form of accidental complexity.
[3] I.E. the caller waits for the callee to execute a RETURN statement.

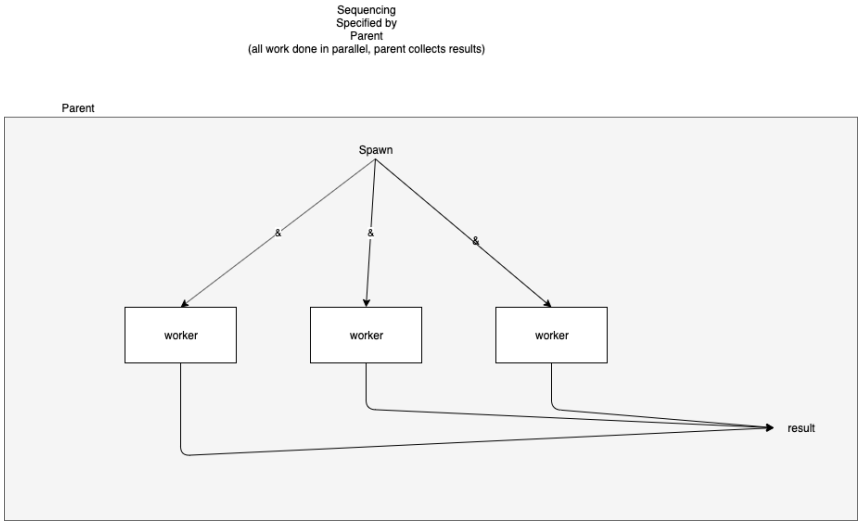
A routine transfers control to another routine and waits for it to complete its processing.

The caller uses a stack - i.e. an optimized collection[4] - to create a list of parameters to the callee, then leaves a breadcrumb (return address) on the stack.

The callee performs processing and leaves a return value (usually a single value) and uses the stacked breadcrumb to return control to the caller.

Stack based sequencing is caused by[5] text-only syntax.

# Spawn and Wait




[4] The stack is an optimized Collection. A stack is an Array allocated in "inexpensive" reusable, memory. An Array is a Collection with most of the housekeeping details optimized away. A Stack is an Array is a Collection with items stored in contiguous locations. Early computer architectures, e.g. some IBM 360s, did not have hardware-supported stacks, and used special instructions, e.g. BALR, to create linked lists of optimized islands of memory. The idea of *scoping* was conflated with *memory and CPU optimization* as was common in early forms of computing.

[5] A form of accidental complexity.

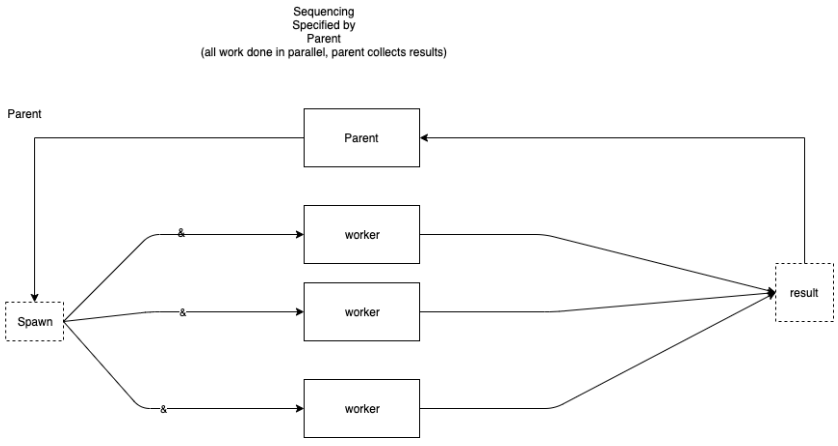
In *spawn and wait* sequencing, the caller "spawns" processes for each child / called routine and then waits until all of the child processes have died (and left results, if any, in distinguished locations).

Spawn and wait has been implemented in several forms:

- UNIX® fork() and waitpid()
- Bash (sh, etc.) & and wait commands
- hardware DMA (Direct Memory Access)
- "par" statement in several "parallel" languages
- node.js (anonymous functions provide *wait()* operation)
- etc.

Spawn and wait has been traditionally conflated with full-blown operating system processes (aka threads). Processes have been traditionally conflated with solutions to the (harder) problems of time-sharing and memory sharing.

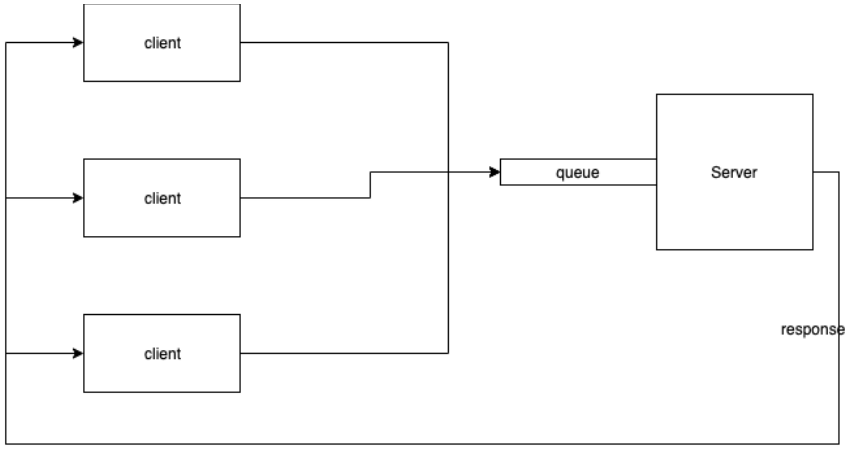
# Spawn and Wait 2




The above diagram is simply another way to diagram a spawn and wait design.

See *spawn and wait*.

# Server

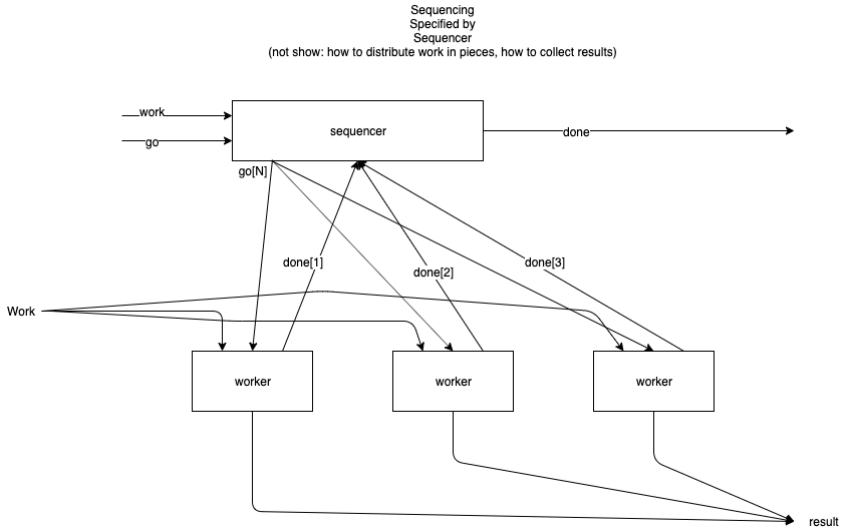



Server-based
Sequencing
(same as Hoare Monitor - one writer)
(e.g. multiple browsers, 1 server)

A *server* based sequencer is one where a single process contains and hides a resource. Client processes send requests for resource data.

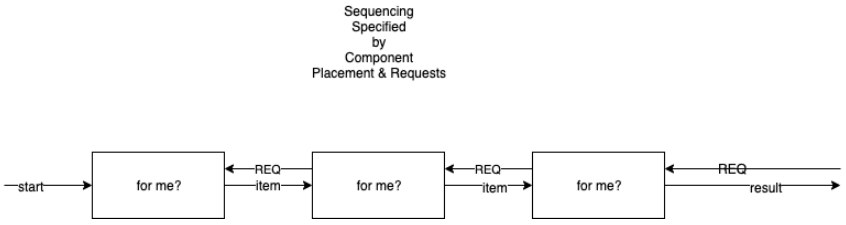
[Note that a Hoare Monitor is a essentially a server-based sequencer in the context of time-sharing and memory sharing. See Hoare Monitor for further discussion.]

# Explicit Sequencer



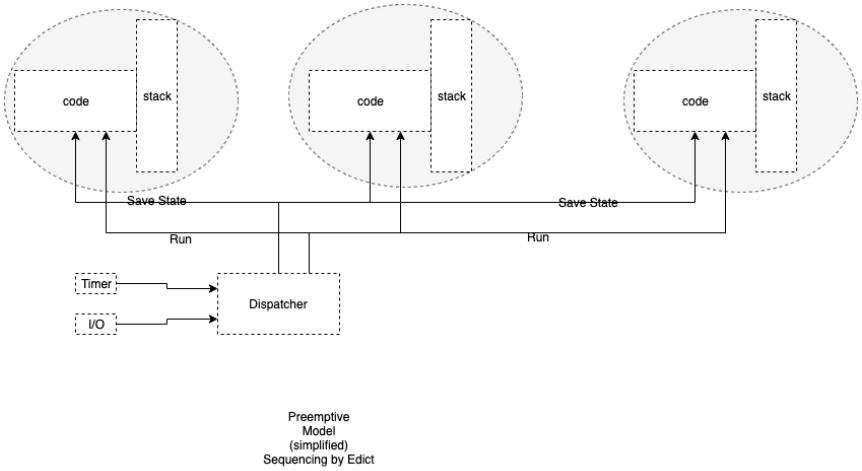

In the Explicit Sequencer design, work is sent to all workers, then a Sequencer process dictates the order in which each worker executes.

# Pull




In a *pull*-based sequence, workers are arranged in a chain and respond to REQuests from downstream components.

# Preemption




Preemptive
Model
(simplified)
Sequencing by Edict

In a *preemption*-based sequence, each worker is given a private memory space and stack. A distinguished routine - the *dispatcher* - determines the order in which workers execute.

The *dispatcher* routine is, typically, supplied by the O/S[6].

# Hoare Monitors

Hoare Monitors operate like the *server* sequencer, but do so in an environment where memory sharing and/or CPU minimization[7] is employed.

A *monitor* is a *server*[8] and all other processes can be clients of the server. "Requests" are made by calling routines that are protected by the monitor. The O/S[9] allows only one process to enter the monitor at a time, and all other requesting processes are *suspended* and placed on a queue, waiting for the monitor to become free.

---

[6] Operating System - essentially a library.

[7] reduced number of CPUs needed - usually as an optimization in the face of costly CPU hardware

[8] The execution thread of the server is provided by the calling processes.

[9] Operating System

Processes inside the monitor could *signal* events to other (waiting) processes.

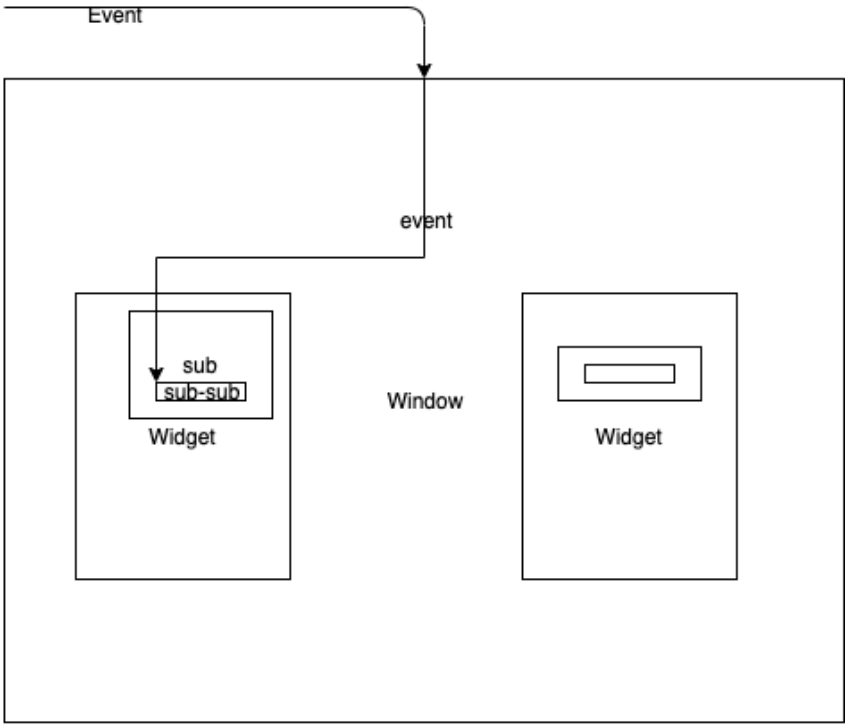
The original manifestation of Hoare Monitors required a *rendezvous* between the process in the monitor and a process waiting on a *monitor signal*.

The requirement for *rendezvous* was relaxed to allow *deferred signals*[10].

See, also, "Server" for a discussion of the basic pattern.

[10] e.g. in the Turing+ language

# Hierarchy

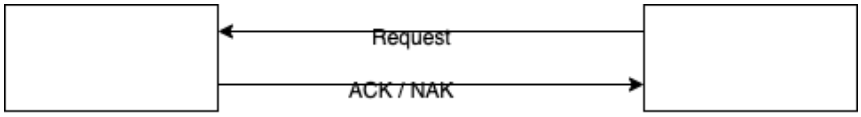



Hierrachical
Sequencing
(e.g. windowing systems)

Hierarchical sequencing is a sequencing style where children components are contained within parents, like Russian dolls. As incoming events arrive, the parent gets first-right-of-refusal to act on the events. If the parent does not act on the events, the events are passed on to contained children, recursively.

This pattern is common in windowing systems.

# Handshake Protocol




Handshake
Sequencing
(e.g. network protocols)

In *handshake* sequencing, a component sends a request to another component (instead of directly "calling" it).

The receiving component responds with a "handshake" message - usually an ACK in the case of successful receipt. The receiver might respond with a NAK (not acknowledge) if it deems that the message was garbled, or, the receiver might not respond at all.

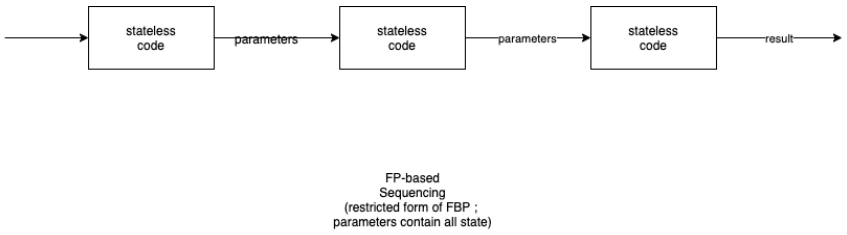
The requestor expects one of 3 responses:

1. ACK - means that the receiver received the message and is acting on it,
2. NAK - means that the receiver received a message, but has deemed that the message has been damaged in transit - the requestor resends the message or, after several retries, declares a send error of some kind,
3. silence - the requestor times-out waiting for an ACK/NAK from the receiver - the requestor resends the message, or, after several retries, declares a send error of some kind.

This *handshake* pattern is most often seen in network protocols.

I would expect to see this pattern arise more often with the advent of (more) distributed computing and IoT.

# Filter Pipelines




Filter pipelines form chains of routines. Each routine in the chain has no side-effects,[11] or its side effects are isolated from the rest of the system.

Information flows strictly down the pipeline, e.g. from left to right. Feedback loops do not exist.

Pipeline sequencing has been implemented in:

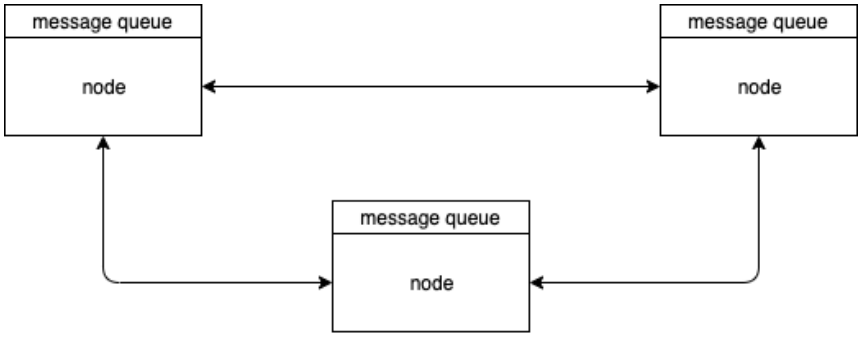
- *Bash* pipelines
- FP filter chains
- Smalltalk ";" operators
- etc.

In a filter pipeline pattern, the data flowing between components contains all of the state.

(See, also, FBP, for a pattern of data flows which allows feedback).

[11] In memory-sharing systems.

# Flat Message Passing

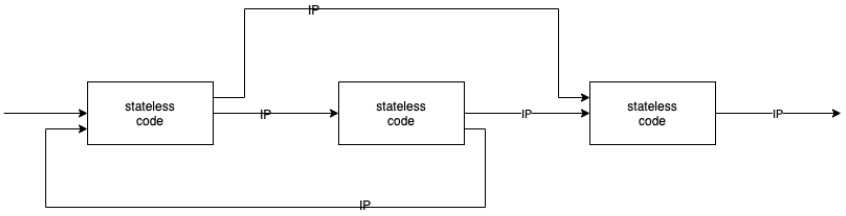



Flat Message Passing
Sequencing

In message passing, every component has an input queue of messages. All components are asynchronous and can process messages at different speeds.

Flat message passing does not scale well to large systems (as is the case for anything that is designed in a flat manner). Flat message passing can be tamed and scaled using hierarchical scoping.

# FBP (Flow Based Programming)




FBP-based
Sequencing
(IPs contains all State)

Each component is a concurrent machine.

Concurrent machines communicate with one another via bounded buffers.

Components have input and output ports that are connected to bounded buffers.

Data flowing between components is called IPs (Information Packets).

Components can read-from and write-to ports in a random manner.

A component suspends if it attempts to send to an output port which has a full buffer.

A component suspends if it attempts to read from an input port that has an empty buffer.

FBP can route IPs in a feedback and feedforward manner.

FBP is similar to FP,[12] in that all state is contained in IPs.

FBP[13] can be used to construct filter pipelines, but FBP is more general in that it allows feedback and feedforward. Unlike pipelines, FBP allows connections that "skip over" components in the chain and connections to components that come "earlier" in the chain.

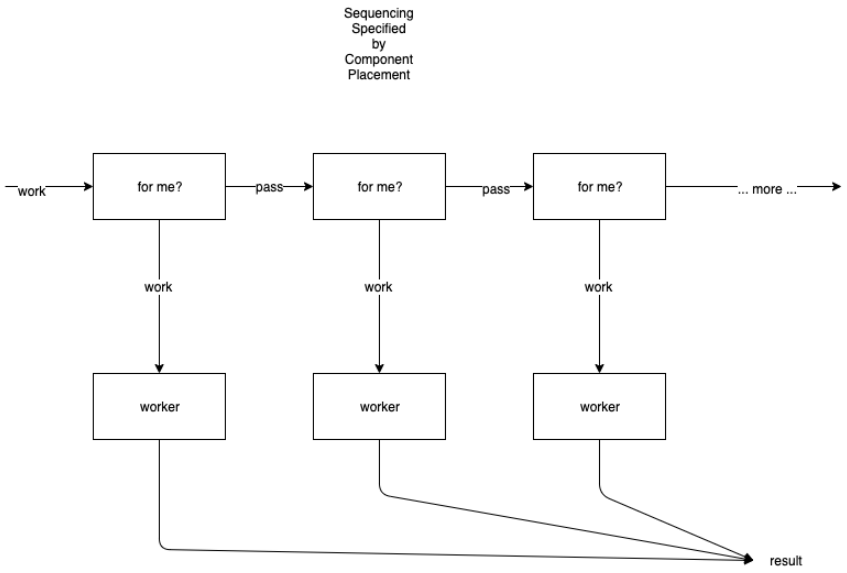
FBP systems have been, traditionally, simulated on top of preemptive sequencing.

(See, also, https://groups.google.com/g/flow-based-programming/c/l5SkeB40iwE)

[12] Functional Programming
[13] Flow-Based Programming

(See also https://jpaulm.github.io/fbp/).

# Daisy Chain




A *daisy chain* sequence arranges components in a chain.

Each component has a unique address.

Each component in the chain inspects incoming messages to determine if the message is addressed to them. If the message is not addressed to the given component, the message is forwarded to downstream components. If the message is addressed to the given component, the message is not forwarded and is processed by the component.

Component outputs are all tied together to form a result. One component processes the message and creates one result (which is fed to the common output).

If no component processes the message, either

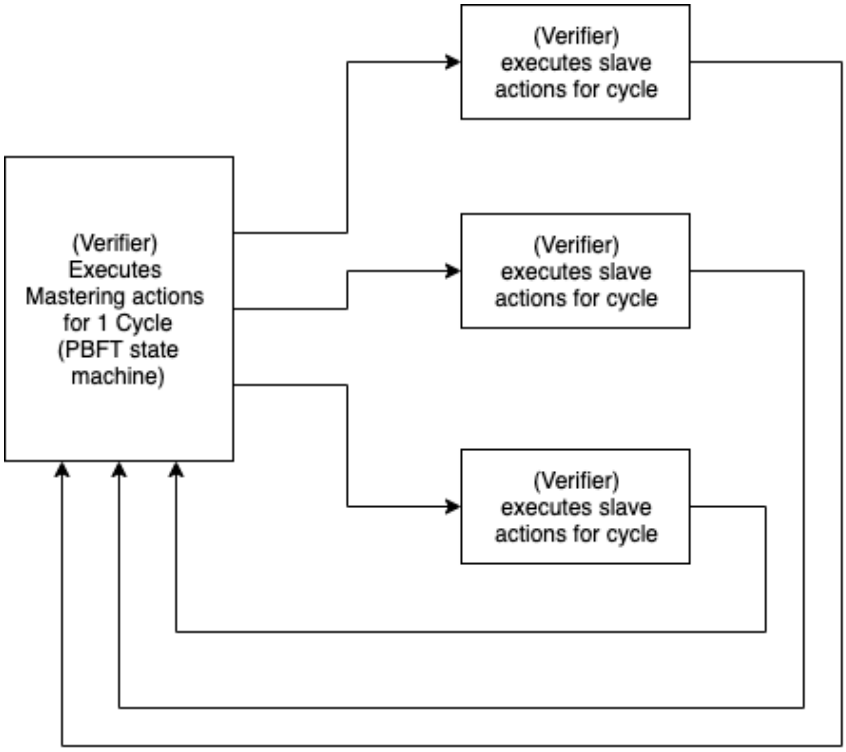
- the message is completely ignored, or,
- the last component in the chain produces some sort of exception result.[14]

A daisy chain system has, logically, two output ports - one is the result, another is an exception.

In a daisy chain, the earlier components in the chain receive priority over later components in the chain. In general, fairness doesn't matter as long as the "work" gets done.

[14] Note that the sender might also be the "last" component in the chain.

# Blockchain



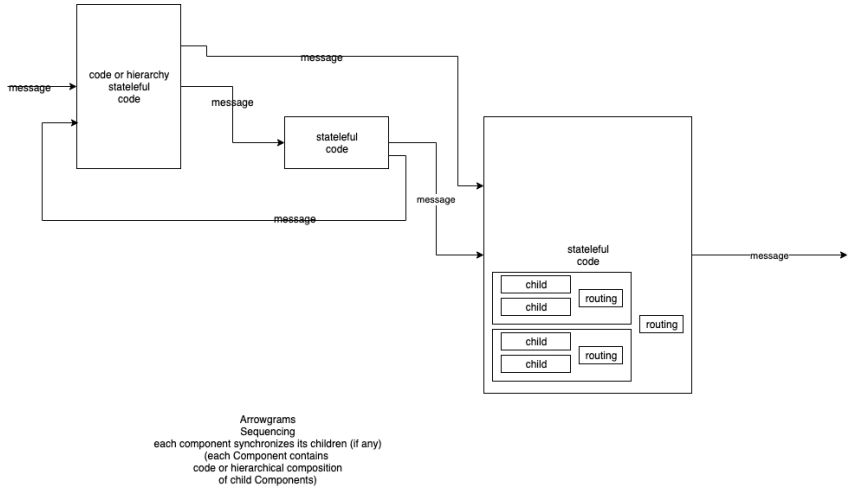

Blockchain
Sequencing
(new master every cycle)
All verifiers are potential
Masters

A blockchain, in current technology, is a spawn-and-wait system wherein the parent node is changed on every "cycle" (defined by the blockchain algorithm). The choice of "parent" is randomized to prevent attackers from guessing the future behaviour of the system.

Blockchains typically consist of two parts:

1. cryptography
2. sequencing.

# Reactive




In the *reactive*[15] pattern, components receive events (aka messages) and react to the messages.

Events (messages) are queued.

One message is processed, fully to completion, before another message is taken from the input queue.

A *reactive* system is a system of concurrent components wherein every component processes events in a hierarchical pattern.

Components are isolated from one another.

Components can be implemented as *composites* or *leaves*.

It is not possible to discern how a component is implemented without looking inside the component.

---

[15] Arrowgrams™ is the trade name of a reactive system that I am developing.

Components have multiple input ports and multiple output ports.

Input events are queued (on a single queue) and a component processes an event to completion before processing another event.

A leaf component processes an event using some other technology, e.g. by using a specific programming language.

A composite component contains children components. A composite component process input events by forwarding them to its children. A composite component is considered "busy" if any of its children are busy.

The *reactive* pattern is a realization of the *divide and conquer* paradigm. A problem can be dissected into two components - *the leaf* and *the rest* (conquer and divide, resp.).