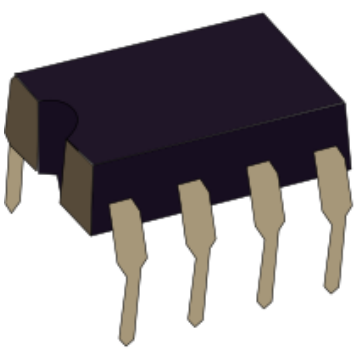


Floyd

FLOYD SYSTEMS MANUAL

Floyd Script is the basic way to create logic. The code is isolated from the noise and troubles of the real world: everything is immutable and pure. Time does not advance. There is no concurrency or communication with other systems, no runtime errors.

Floyd Systems is how you make software that lives in the real world, where all those things happens all the time. Floyd allows you create huge, robust software systems that you can reason about, spanning computers and processes, handling communication and time advancing and faults.

Floyd uses the C4 model to organize all of this. C4 model https://c4model.com/

GOALS

1. A high-level way to organise huge code bases and systems with many threads, processes and computers, beyond functions, classes and modules and also represent those concepts through out: at the code level, in debugger, in profiler etc.
2. A simple and robust method for doing concurrency, communication etc.
3. Allow extreme performance and profiling as a separate thing done ONTOP of the correct logic.
4. Support nextgen visual programming and interactions.

ABOUT PERFORMANCE

Floyd is designed to make it practical to make big systems with performance better than what you get with average optimized C code.

It does this by splitting the design into two different concepts:

1. Encourage your logic and processing code to be simple and correct and to declare where there is opportunity to execute code independely of eachother.

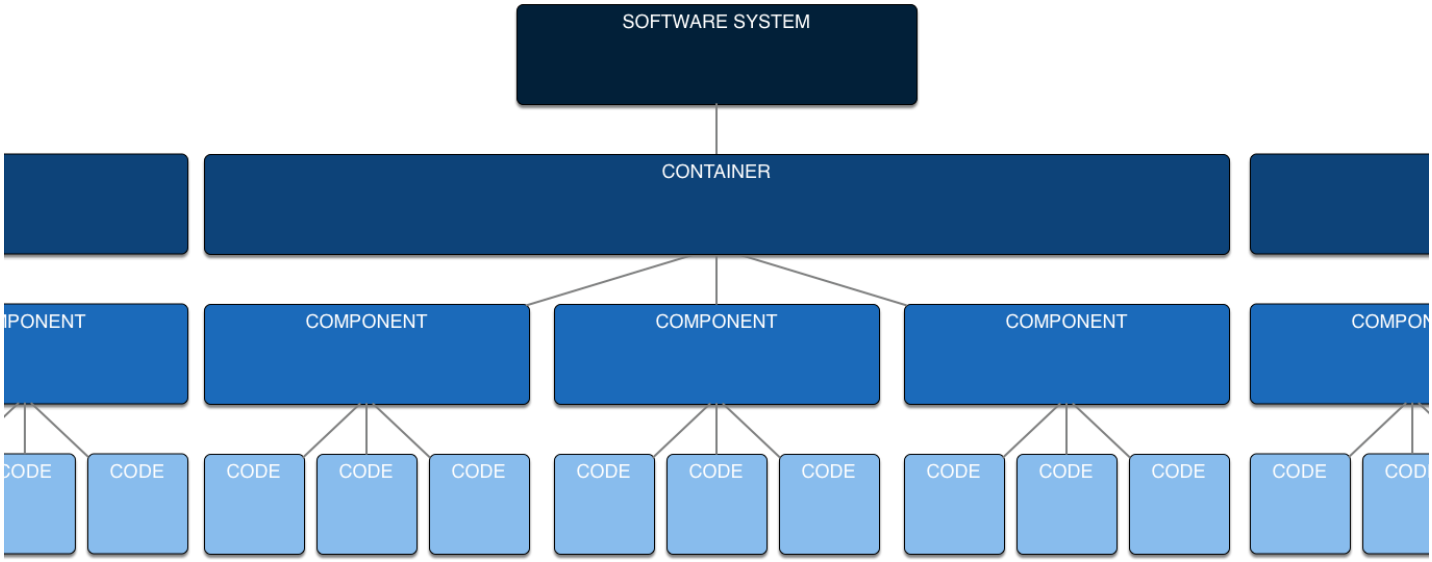
2. At the top level, profile execution and make high-level improvements that dramatically alter how the

code is *generated* and executed to run on the available hardware. Caching things, working in batches, running in parallel, ordering work for different localities, memory layouts and access patterns.

It is also simple to introduce concurrency to create more opportunities to run computations in parallel.

ABOUT C4 CONCEPTS

A complete system is organised into parts like this:



PERSON

Various human users of your software system

SOFTWARE SYSTEM

Highest level of abstraction and describes something that delivers value to its users, whether they are human or not.

Floyd file: **software-system.floyd**

CONTAINER

A container represents something that hosts code or data. A container is something that needs to be running in order for the overall software system to work. Mobile app, Server-side web application, Client-side web application, a microservice are all examples of containers.

This is usually a single OS-process, with mutation, time, several threads. It looks for resources and knows how to string things together inside the container.

The container wires together a bunch of unpure components.

COMPONENT

Grouping of related functionality encapsulated behind a well-defined interface. Like a software integrated circuit or a code library. Does not span processes. JPEG Librart, JSON lib. Custom component for syncing with your server. Amazon S3 library, socket library.

There are two types of components: pure and unpure.

- Pure components have no side effects, like a ZIP library or a matrix-math library. They are also passive.
- Unpure components may be active (detecting mouse clicks and calling your code) and may affect the world around you or give different results for each call. get*time() and get*mouse*pos(), on*mouse*click(). read*directory_elements().

CODE

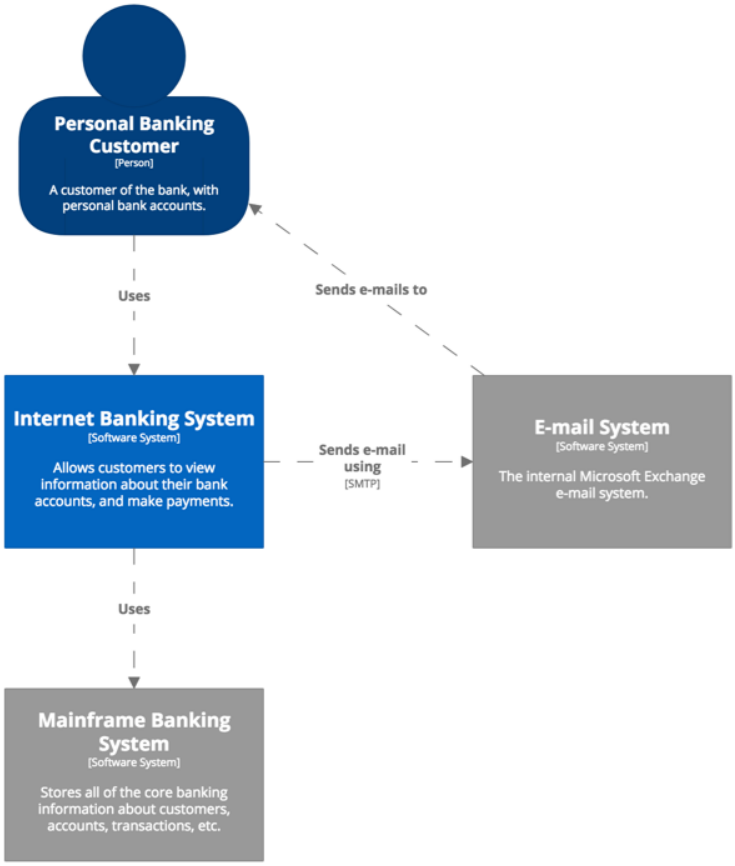
Classes. Instance diagram. Examples. Passive. Pure.

```
jpeg_quantizer.floyd, colortab.floyd -- implementation source files for the jpeglib
```

ABOUT C4 DIAGRAMS

There is a set of standard diagram views for explaining and reasoning about your software:

```
<img src="floyd_systems_level1_diagram.png" alt="drawing" width="600px"/>
```

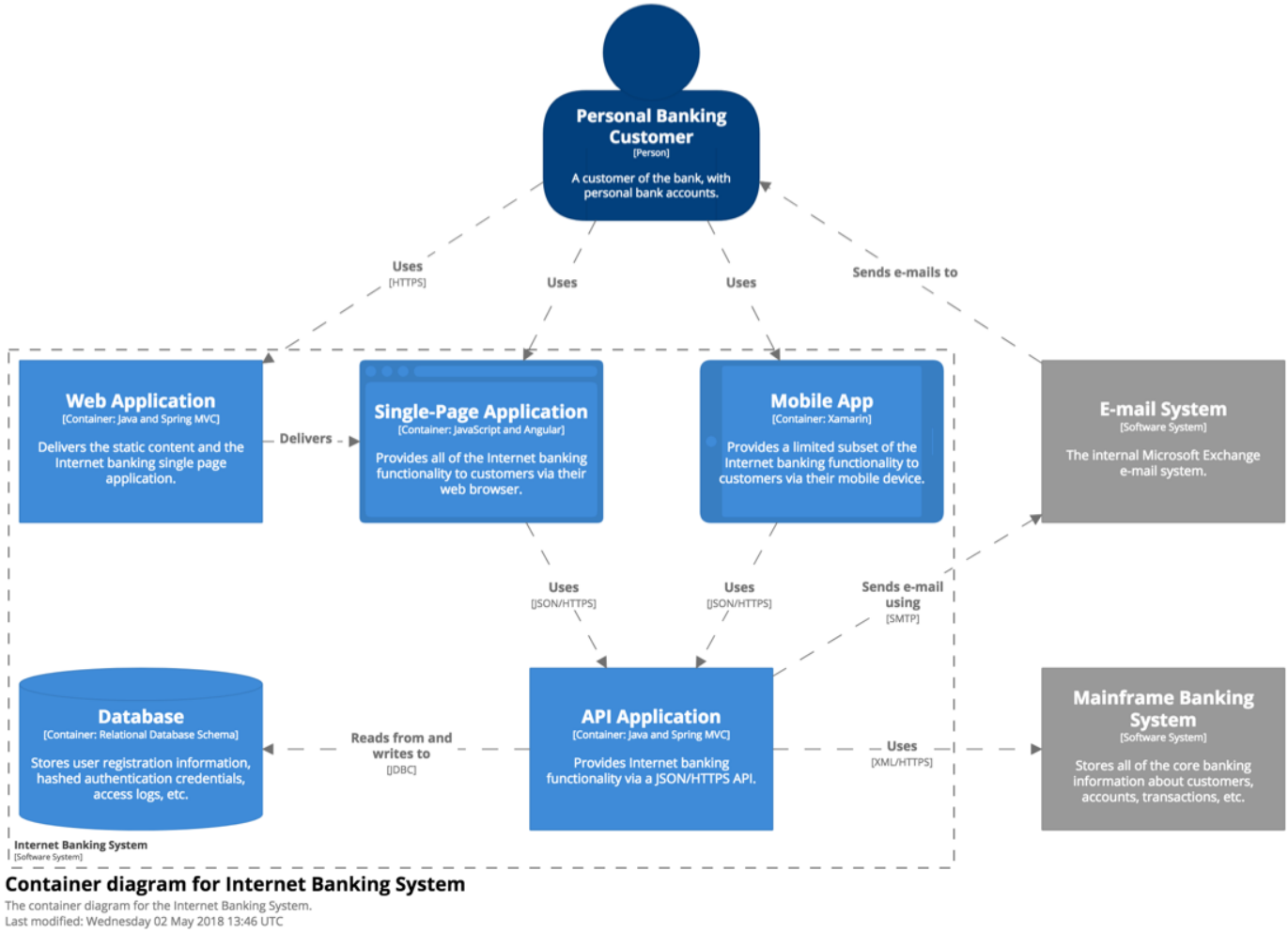


System Context diagram for Internet Banking System

The system context diagram for the Internet Banking System.
Last modified: Wednesday 02 May 2018 13:46 UTC

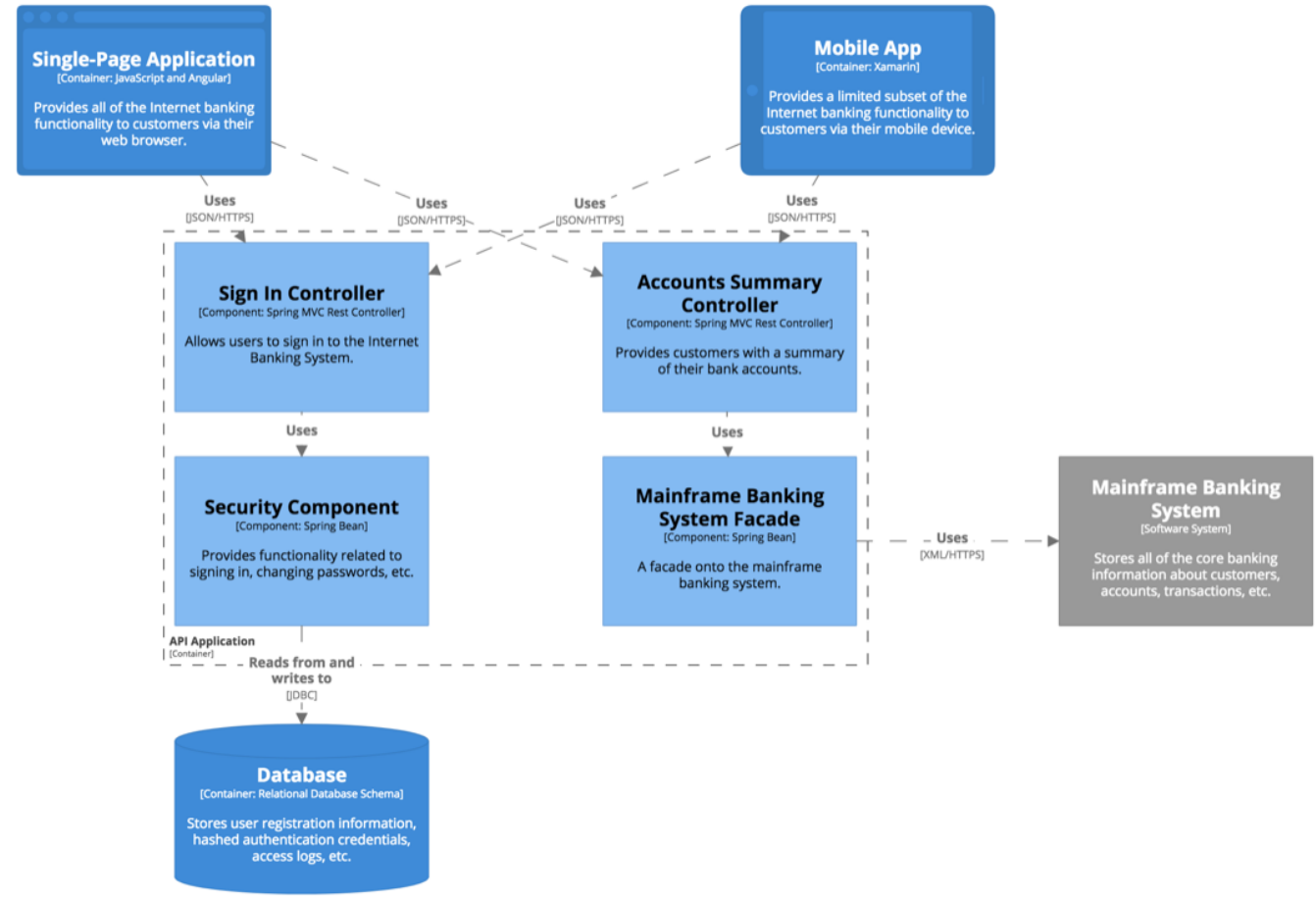
Above, level 1: System Context diagram

```
<img src="floyd_systems_level2_diagram.png" alt="drawing" width="600px"/>
```



Above, level 2: Container diagram

```
<img src="floyd_systems_level3_diagram.png" alt="drawing" width="600px"/>
```

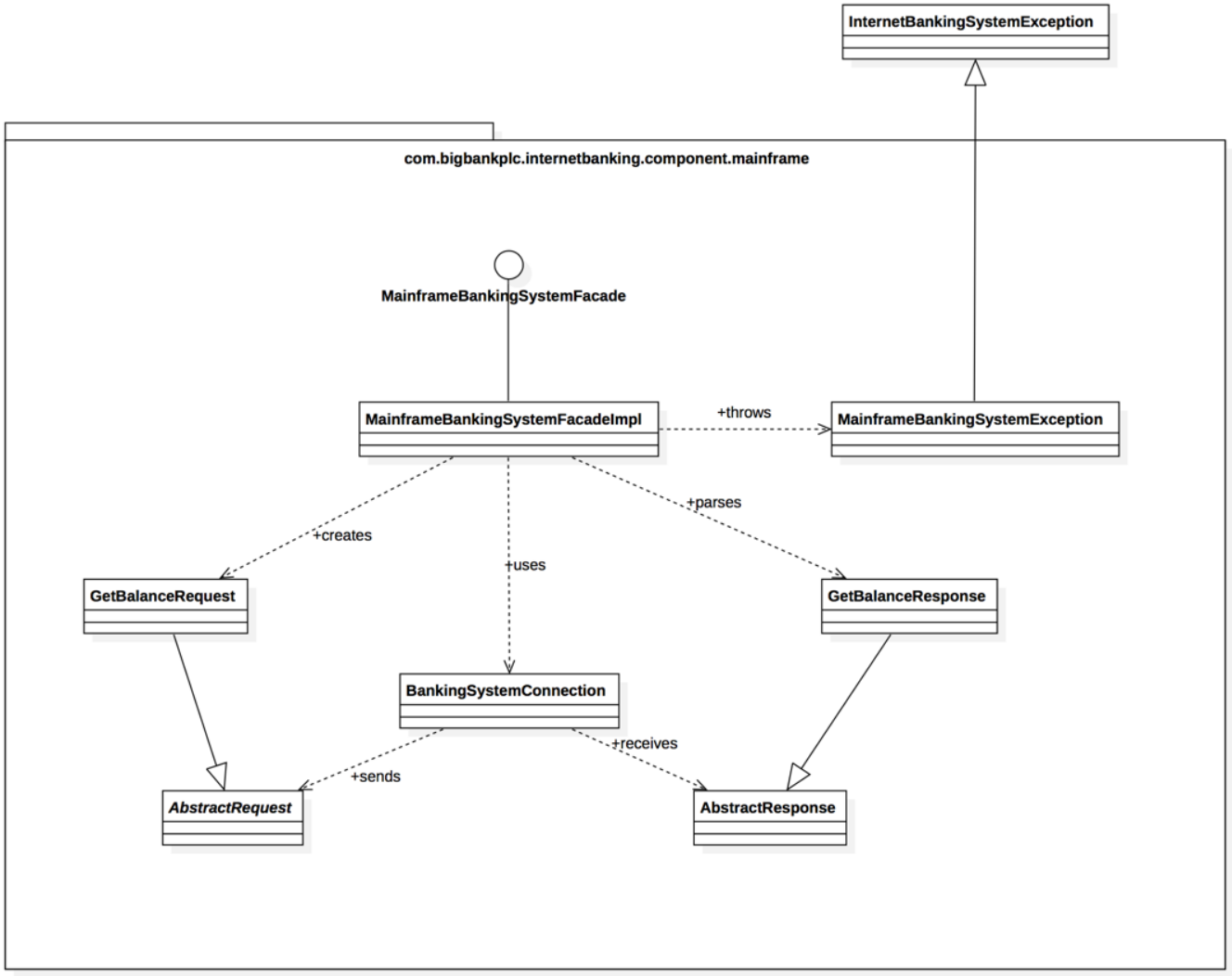


Component diagram for Internet Banking System - API Application

The component diagram for the API Application.
Last modified: Wednesday 02 May 2018 13:46 UTC

Above, level 3: Component diagram

```
<img src="floyd_systems_level4_diagram.png" alt="drawing" width="600px"/>
```



Above, level 4: Code

Notice: a component used in several containers or a piece of code that appears in several components will appears in each, seamingly like duplicates. The perspective is **logic dependencies**. There is no diagram for showing the physical dependencies -- which source files or libraries that depends on eachother.

ABOUT CONTAINERS

Containers are how you make a mobile app or server. By writing code, stringing together existing components and deciding how to relate to the world around the container. There are often other, sibling containers in your system, for example a server for your mobile game. Containers may be implemented some other way but should still be represented in Floyd Systems, in this case by using a proxy.

The container connects components together using wires using a declaration file. Wires carry messages, which is a Floyd value and typically an enum. Notice that the message can carry *any* of Floyd's types -- even huge multi-gigabyte nested structs or collections. Sincce they are immutable they can be passed around efficiently (internally this is done using their ID or hardware address).

The container completely defines: concurrency, state, communication with ourside world and runtime errors of the container. This includes sockets, file systems, messages, screens, UI.

When you design a container the focus is on the unpure components. Components on which your used components depend upon are automatically shown too, if they are unpure. This makes *all* unpure components visible in the same view -- no unpureness goes on anywhere out of sight.

Probes and tweakers are added ontop of a design. They allow you to augument a design with logging, profiling, breakpoints and do advanced performance optimizations, all without altering the code or design itself. The tweakers cannot affect the behaviour of the logic, only timing and hardware utilisation etc.

A container is typically run as its own OS process.

??? Give separate names to unpure component vs pure component. Only unpure components are important when designing the container.

Example of components you drop into your container:

- Actor component: an unure component written in Floyd Script that has its own state & inbox.
- Multiplexer componenet: contains internal pool of clocks and distributes incoming messages to them to run messages in parallel.
- Built in local FS component: Read and write files, rename directory, swap temp files
- Built in S3 component
- Built in socket component
- Built in REST-component
- Built in UI-component
- Built in command line component: Interfaces with command line arguments / returns.
- Audio component that uses Direct X
- VST-plugin component
- A component written in C

Notice: these components are all non-singletons - you can make many instances in one container

??? IDEA: Glue expressions, calling FLoyd functions ??? What if you want a container-like setup for each "document"? Allow making sub-container that can be instantiated in a container? Or a tree of stuff inside a container.

NON-GOALS

- Be reusable.
- To be composable
- Pure / free of side effects

ABOUT CONCURRENCY AND TIME IN DETAIL

This section describes how to express time / mutation / concurrency in Floyd. These things are related and they are all setup at the top level of the container. This is the main **purpose** of the container.

The goal with Floyd's concurrency model is to be:

1. Simple and robust pre-made mechanisms for real-world concurrency need. Avoid general-purpose primitives.
2. Composable.
3. Allow you to make a static design of your container and its concurrency and state.
4. Separate out parallelism into a separate mechanism.
5. Avoid problematic constructs like threads, locks, callback hell, nested futures and await/async -- dead ends of concurrency.
6. Let you control/tune how many threads and cores to use for what parts of the system, independantly of the code.

Inspirations for Floyd's concurrency model are CSP, Erlang, Go routies and channels and Clojure Core.Async.

ACTORS: INBOX, STATE, FUNCTION

For each independant state or "thread" you want in your container, you need to insert an Actor component. You need to write its processing function and define its mutable state. An actor receives messages via its inbox. The inbox is threadsafe and it's THE way to communicate across actors.

The actor represents a little standalone process that listens to messages from other actors. Those actors may run in the same or other hardware threads or green threads or even synchronously.

When an actor receieves a message in its inbox, its function is called (now or soon) with that message and the actor's previous state. The actor does some work - something simple or a maybe call a big call tree and it ends by returning its new state, which completes the message handling.

The actor function may be called synchronously when client posts it a message, or it may be run on a green thread, a hardware thread etc. It may be run the next hardware cycle or at some later time.

Actors is the only way to keep state in Floyd.

Insight: synchronization points between systems (state or concurrent) always breaks composition. Move these to top level of container.

The inbox has two purposes:

1. Allow component to *wait* for external messages using the select() call.

2. Move messages between different clock-bases -- they are thread safe and atomic.

You can always post a message to *any* clock-component, even when it runs on another clock.

A clock is statically instantiated in a container -- you cannot allocate them at runtime.

The actor function CAN chose to have several select()-statements which makes it work as a small state machine.

The actors cannot change any other state than its own, excep sending messages to other actors or call unpure functions. A clock can send messages to other clocks, optionally blocking on a reply, handling replies via its inbox or not use replies.

The clock's function is unpure. The function can call OS-function, block on writes to disk, use sockets etc. Clocks needs to take all runtimes they require as arguments. Clock function also gets input token with access rights to systems.

The runtime can chose to place clocks on different cores and processors or servers. You can have several clocks running in parallel, even on separate hardware. These forms separate clock circuits that are independent of eachother. You have control over this via tweakers.

Clocks are inexpensive, you can use 100.000-ands of clocks. Clocks typically runs as M x N threads. They may be run from the main thread, a thread team or cooperatively.

Who decides when to advances the clocks? Runtime advances a clock when it has at least one message in its inbox. Runtime settings controls how runtime prioritizes the clocks when many have waiting messages.

An actor can be synced to another actor's clock. All posts to the inbox will then be synchrnous and blocking calls. These types of clock still have their own state and can be used as controllers / mediators -- even when it doesnt need its own thread.

ACTOR LIMITATIONS:

- Cannot find assets / ports / resources — those are handed to it via the container's wiring.
- Cannot go find resources, processes etc. These must be handed to it.
- Clocks cannot be created or deleted at runtime.
- Cannot create other clocks!
- ??? IDEA: Supervisors: this is a function that creates and tracks and restarts clocks.
- ??? System provices clocks for timers, ui-inputs etc. When setup, these receive user input messages etc.

GAIN PERFORMANCE VIA CONCURRENCY

Sometimes we introduce concurreny to make more parallelism possible: multithreading a game engine is taking a non-concurrent design and making it concurrent to be able to improve throughput by running many tasks in parallel. This is different to using concurrency to model real-world concurrency like UI vs background cloud com vs realtime audio processing.

CONCURRENCY SCENARIOS

#	Need	Traditional	Floyd
1	Make a REST request	Block entire thread / nested callbacks / futures / async-await	Just block. Make call from Actor to keep caller running
2	Make a sequence of back and forths with a REST server	Make separate thread and block on each step then notify main thread on completion / nested futures or callbacks / await-async	Make an Actor that makes blocking calls
3	Perform non-blocking unpure background calculation (auto save doc)	Copy document, create worker thread	Use actor, use data directly
4	Run process concurrently, like analyze game world to prefetch assets	Manually synchronize all shared data, use separate thread	Use actor -- data is immutable
5	Handle requests from OS quickly, like call to audio buffer switch process()	Use callback function	Use actor and set its clock to sync to clock of buffer switch
6	Improve performance using concurrency + parallelism / fan-in-fan-out / processing pipeline	Split work into small tasks that are independant, queue them to a thread team, resolve dependencies some how, use end-fence with completetionnotification	call map() or supermap() from an Actor.
7	Spread heavy work across time (do some processing each game frame)	Use coroutine or thread that sleeps after doing some work. Wake it next frame.	Actor does work. It calls select() inside a loop to wait on next trigger to continue work.
8	Do work regularly, independant of other threads (like a timer interrupt)	Call timer with callback / make thread that sleeps on event	Use Actor that calls post*at*time(now() + 100) to itself
9	Small server	Write loop that listens to socket	Use Actor that waits for messages

ABOUT PARALLELISM

In floyd you accellerate the performance of your code by making it expose where there are dependencies between computations and where there are not. Then you can orcestrate how to best execute your container from the top level -- using tweak probes and profiling probes, affecting how the hardware is mapped to your logic.

Easy ways to expose parlellism is by writing pure functions (their results can be cached or precomputed) and by using functions like map(), fold(), filter() and supermap(). These function work on individual elements of a collection and each computation is independant of the others. This lets the runtime process the different elements on parallel hardware.

map() processes each element in a collection using a function and returns a new collection with the results. supermap() works like map(), but each element also has dependencies to other elements in the collection.

Accelerating computations (parallelism) is done using tweaks — a separate mechanism. It supports moving computations in time (lazy, eager, caching) and running work in parallel.

Often actors and concurrency is introduced into a system to *expose opportunity* for parallelism.

The optimizations using tweaks in no way affect the logic of your program, only the timing and order where those don't matter.

To make something like a software graphics shaders, you would do

let image2 = map(image1, my*pixel*shader) and the pixels can be processed in parallel.

Task - this is a work item that takes usually approx 0.5 - 10 ms to execute and has an end. The runtime generates these when it wants to run map() elements in parallel. All tasks in the entire container are schedueled together.

EXAMPLE SETUPS FOR SOME APPLICATIONS

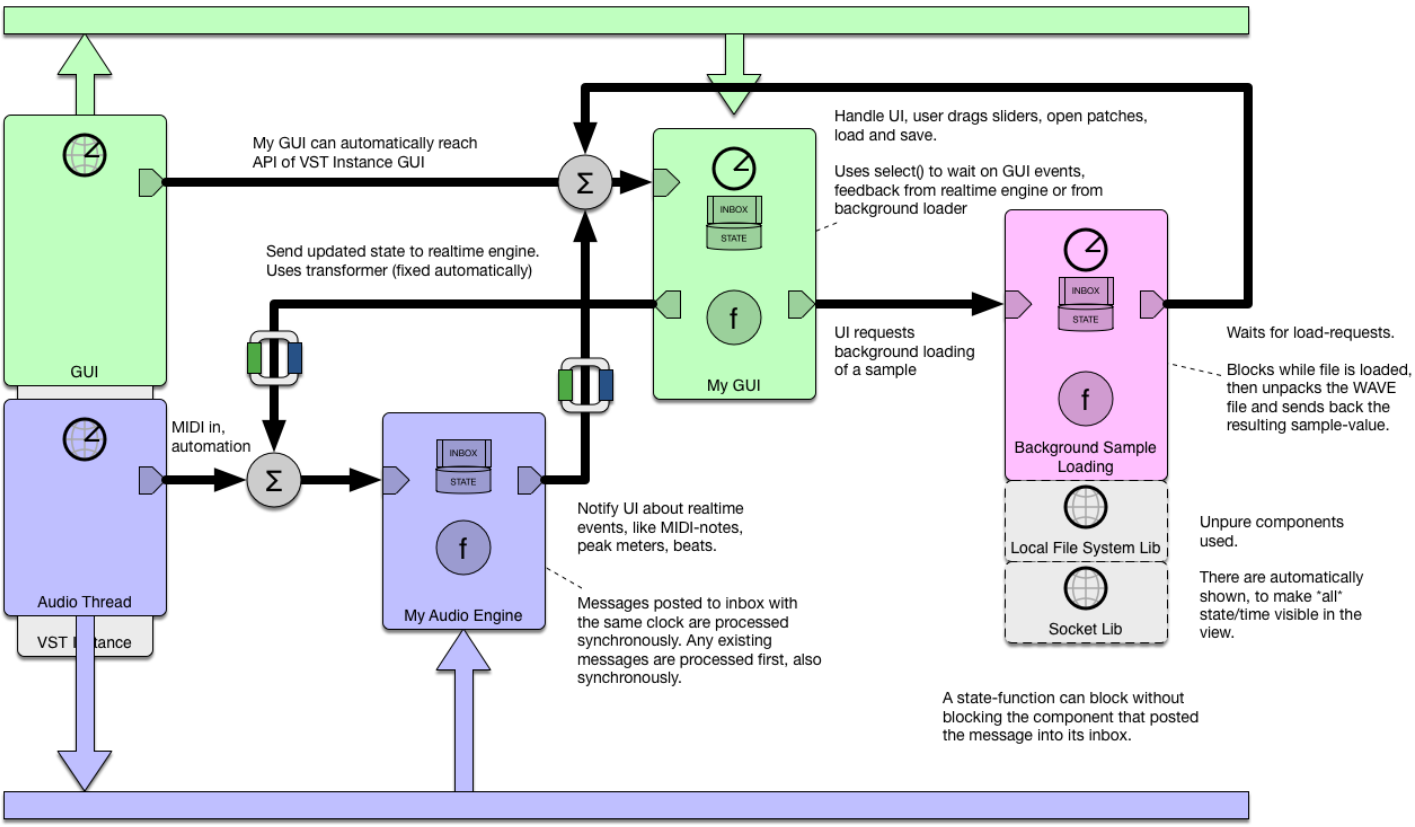
SIMPLE CONSOLE PROGRAM

??? TBD

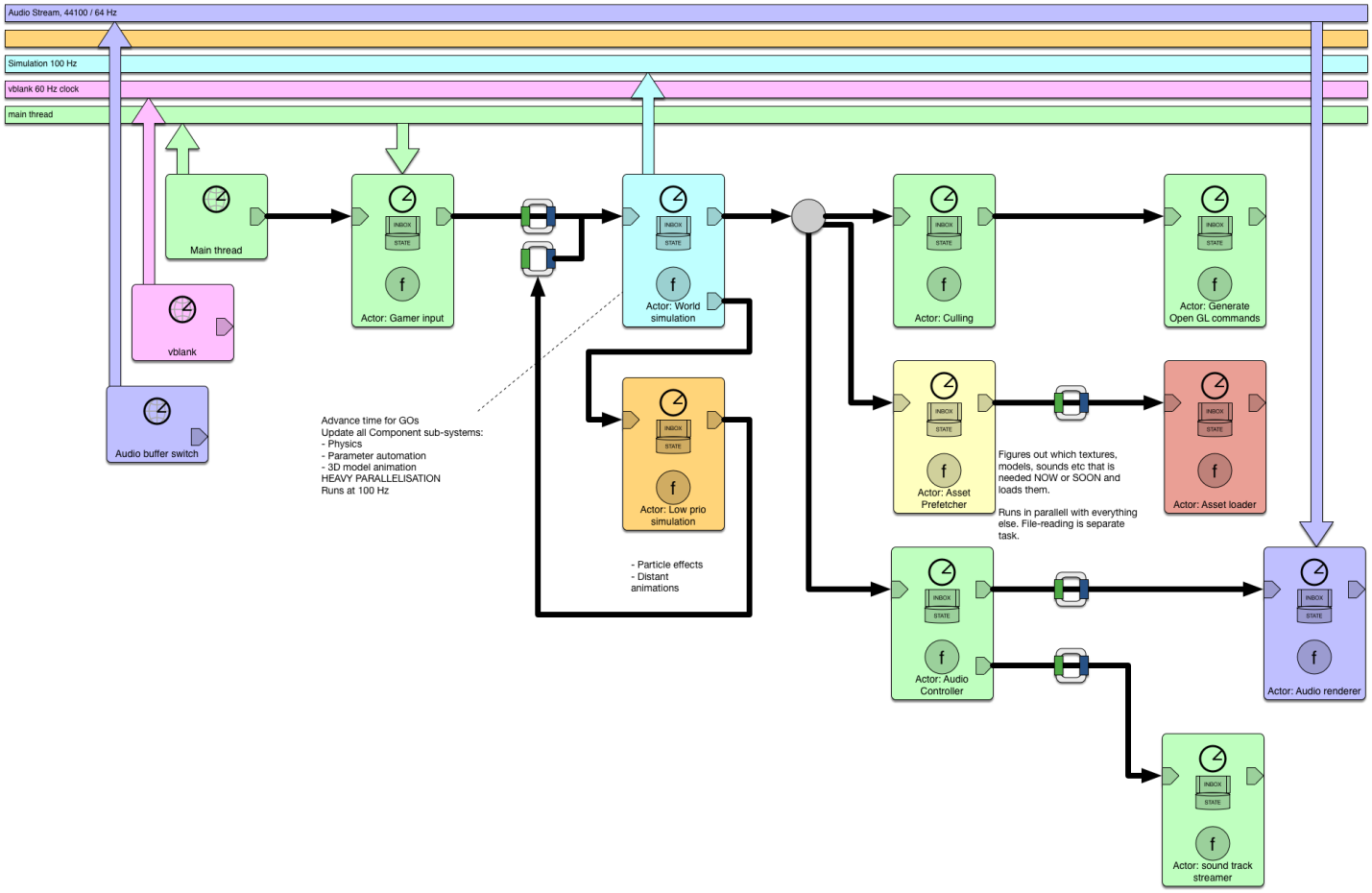
This is a basic command line app, have only one clock that gathers ONE input value from the command line arguments, calls some pure Floyd Script functions on the arguments, reads and writes to the world, then finally return an integer result. A server app may have a lot more concurrency. main() one clock only.

EXAMPLE: VST-plugin

??? example Software System diagram and other diagrams.



FIRST PERSON SHOOTER GAME



https://www.youtube.com/watch?v=v2Q_zHG3vqg

A video game may have several clocks:

- UI event loop clock
- Prefetch assets clock
- World-simulation / physics clock
- Rendering pass 1 clock
- Commit to OpenGL clock
- Audio streaming clock

This game does audio and Open GL graphics. It runs many different clocks. It uses supermap() to render Open GL commands in parallel.

Instagram app

??? TBD

- main ui thread()
- rendering / scaling clock
- server comm clock

??? IDEA: Assigning things to physical threads -- algorithmically

FLOYD SYSTEM REFERENCE

CONTAINER FILE FORMAT REFERENCE

helloworld.container This is a declarative file that describes the top-level structure of an app. Its contents looks like this:

```
    my first design.container

    {
        "major_version": 1,
        "minior_version": 0,

        "nodes": {
        }
    }
```

CONTAINER MAIN REFERENCE

Top level function

```
container_main()
```

This is the container's start function. It will find, create and and connect all resources, runtimes and other dependencies to boot up the container and to do executibe decisions and balancing for the container.

COMMAND LINE COMPONENT REFERENCE

??? TBD

```
int on_commandline_input(string args) unpure
void print(string text) unpure
string readline() unpure
```

LOG PROBE REFERENCE

??? TBD

- Pulse everytime a function is called
- Pulse everytime a clock ticks
- Record value of all clocks at all time, including process PC. Oscilloscope & log

PROFILE PROBE REFERENCE

CACHE TWEAKER REFERENCE

??? TBD

A cache will store the result of a previous computation and if a future computation uses the same function and same inputs, the execution is skipped and the previous value is returned directly.

A cache is always a shortcut for a (pure) function. You can decide if the cache works for *every* invocation of a function or limit the cache to invocations of the function within specified parent part.

EAGER TWEAKER REFERENCE

??? TBD

Like a cache, but calculates its values *before* clients call the function. It can be used to create a static lookup table at app startup.

BATCH TWEAKER REFERENCE

??? TBD

When a function is called, this part calls the function with similar parameters while the functions instructions and its data sits in the CPU caches.

You supply a function that takes the parameters and make variations of them.

LAZY TWEAKER REFERENCE

??? TBD

Make the function return a future and don't calculate the real value until client accesses it.

WATCHDOG PROBE REFERENCE

??? TBD

BREAKPOINT PROBE REFERENCE

??? TBD

TWEAKS - OPTIMIZATIONS REFERENCE

??? TBD

- Insert read cache
- Insert write cache
- Precalculate / prefetch, eager
- Lazy-buffer
- Content de-duplication
- Cache in local file system
- Cache on Amazon S3
- Parallelize pure function
- Increase mutability
- Increase random access speed
- Increase forward read speed, stride
- Increase backward read speed, stride
- Rearrange nested composite (turn vec to struct{ vec, vec, vec }
- Batching: make 64 value each time?
- Speculative batching with rewind.

MAP FUNCTION REFERENCE

map(), fold() filter()

Expose possible parallelism of pure function, like shaders, at the code level (not declarative). The supplied function must be pure.

??? make pipeline part. https://blog.golang.org/pipelines

The functions map() and supermap() replaces FAN-IN-FAN-OUT-mechanisms.

You can inspect in code and visually how the elements are distributed as tasks.

SUPERMAP FUNCTION REFERENCE

```
[int:R] supermap(tasks: [T, [int], f: R (T, [R]))
```

This function runs a bunch of tasks with dependencies between them and waits for them all to complete.

- Tasks can block.
- Tasks cannot generate new tasks. A task *can* call supermap.

Notice: supermap() shares threads with other mechanisms in the Floyd runtime. This mean that even if your tasks cannot be distributed to all execution units, other things going on can fill those execution gaps with other work.

- **tasks**: a vector of tasks and their dependencies. A task is a value of type T. T can be an enum to allow mixing different types of tasks. Each task also has a vector of integers tell which other tasks it depends upon. The task will not be executed until those tasks have been run. The indexes are the indexes into the tasks-vector.

- **f**: this is your function that processes one T and returns a result R. The function must not depend on the order in which tasks execute. When f is called, all the tasks dependency tasks have already been executed and you get their results in [R].

- **result**: a vector with one element for each element in the tasks-argument. The order of the elements are undefined. The int specifies which task, the R is its result.

When supermap() returns all tasks have been completed.

Notice: your function f can send messages to a clock — this means another clock can start consuming results while supermap() is still running.

??? IDEA: Make this a two-step process. First analyse tasks into an execution description. Then use that description to run the tasks.

This lets you keep the execution description for next time, if tasks are the same.

Also lets you inspect the execution description & improve it or create one for scratch.

- If IO is bottleneck, try to spread out IO over time. If IO blocks is bottleneck, try to start IO ASAP.

- Try to keep instructions and data in CPU caches.

??? Have pipeline-part instead of supermap()? ??? No, supermap() is a better solution.

This allows you to configure a number of steps with queues between them. You supply a function for each step. All settings can be altered via UI or programatically.

Notice: supermap() has a fence at end. If you do a game pipeline you can spill things with proper dependencies over the fences.

THE FILE SYSTEM PART

??? TBD All file system functions are blocking. If you want to do something else while waiting, run the file system calls in a separate clock. There are no futures, callbacks, async / await etc. It is very easy to write a function that does lots of processing and conditional file system calls etc and run it concurrently with other things.

??? Simple file API file*handle open*fileread(string path) unpure file*handle make*file(string path) unpure void close*file(file*handle h) unpure

```
vec<ubyte> v = readfile(file_handle h, int start = 0, int size = 100) unpure
delete_fsnode(string path) unpure

make_dir(string path, string name) unpure
make_file(string path, string name) unpure
```

THE REST-API PART

??? TBD

THE S3 BLOB PART

??? TBD

Use to save a value to local file system efficiently. Only diffs are stored / loaded. Allows cross-session persistance. Uses SHA1 and content deduplication.

THE LOCAL FS BLOB PART

??? TBD

Use to save a value to local file system efficiently. Only diffs are stored / loaded. Allows cross-session persistance. Uses SHA1 and content deduplication.

THE ABI PART

??? TBD

This is a way to create a component from the C language, using the C ABI.

STATIC DEFINITION

??? TBD

```
    struct clock_xyz_state_t {
        int timestamp
        [xyz_record_t] recs
    }

    struct clock_xyz_message_st {
        int timestamp
        int mouse_x
        int mouse_y
        case stop: struct { int duration }
        case on_mouse_move:
        case on_click: struct { int button_index }
    }
    clock_xyz_state_t tick_clock_xyz([clock_xyz_state_t] history, clock_xyz_messag
e_st message){
    }
```

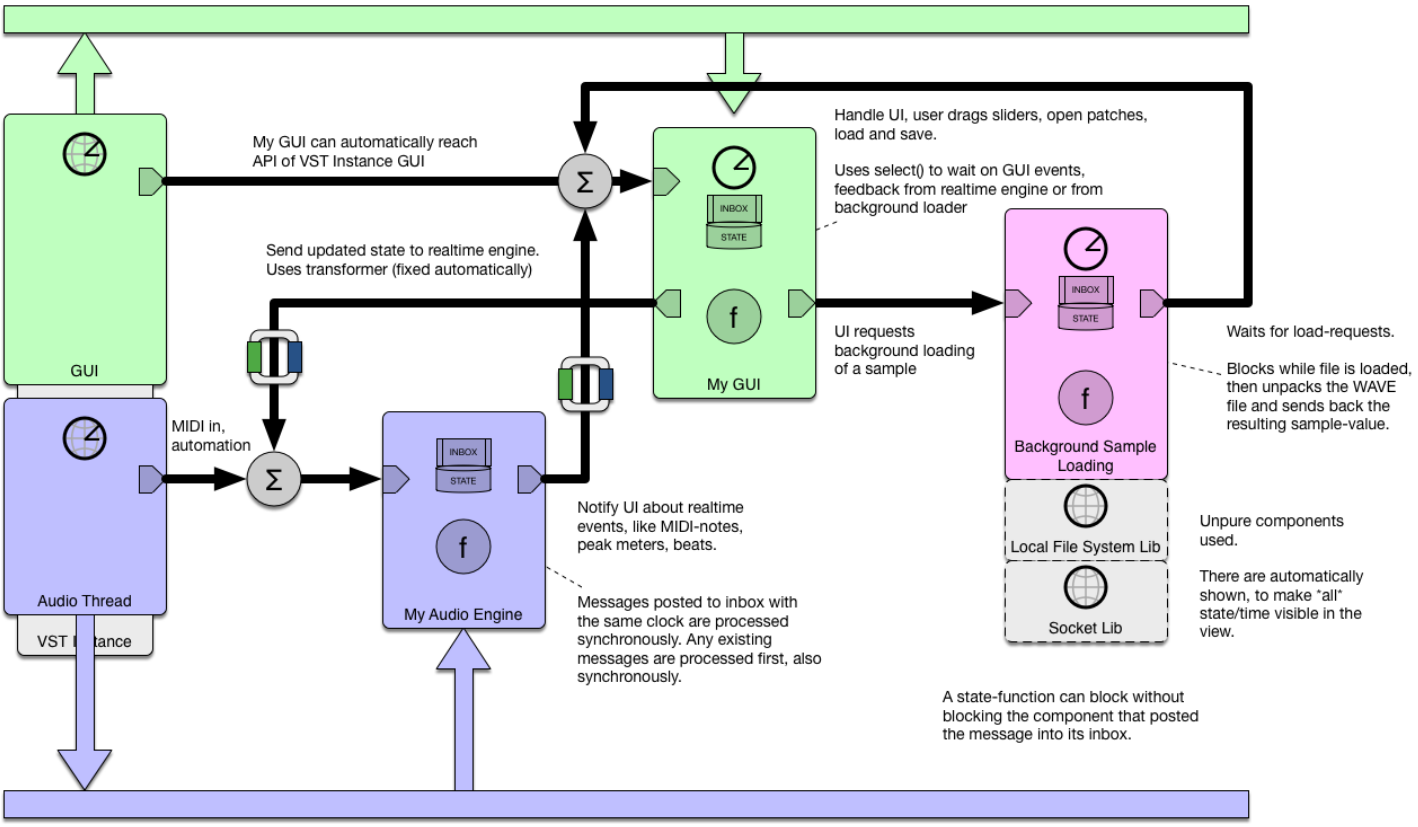
EXAMPLE ACTOR

??? TBD

```
    defclock mytype1_t myclock1(mytype1_t prev, read_transformer<mytype2_t> transf
orm_a, write_transformer<mytype2_t> transform_b){
        ... init stuff.
        ... state is local variable.

        let a = readfile()  //  blocking

        while(true){
            let b = transform_a.read()  //  Will sample / block,  depending on tra
nsformer mode.
            let c = a * b
            transform_b.store(c)    //  Will push on tranformer / block, depending
 on transformer mode.

            //  Runs each task in parallel and returns when all are complete. Retu
rns a vector of values of type T.
            //  Use return to finish a task with a value. Use break to finish and
abort any other non-complete tasks.
            //??? Split to parallel_tasks_race() and parallel_tasks_all().
            let d = parallel_tasks<T>(timeout: 10000) {
                1:
                    sleep(3)
                    return 2
                2:
                    sleep(4)
                    break 100
                3:
                    sleep(1)
                    return 99
            }
            let e = map_parallel(seq(i: 0..<1000)){ return $0 * 2 }
            let f = fold_parallel(0, seq(i: 0..<1000)){ return $0 * 2 }
            let g = filter_parallel(seq(i: 0..<1000)){ return $0 == 1 }
        }
    }
```

EXAMPLE: VST-plugin

??? example Software System diagram and other diagrams.



Source file: *my_vst_plug.container*

```
        //  IMPORTED FROM GUI
        struct uievent {
            time timestamp;
            variant<>
                struct {
                    int x;
                    int y;
                    int mouse_button
                    uint32 mods
                } click;
                struct {
                    int keycode;
                    int x;
                    int y;
                    uint32 mods;
                } key;
                struct {
                    rect dirty
                    channel<obuf> reply
                } draw;
                struct {} activate;
                struct {} deactivate;
            } type;
        }
```

```
struct ibuf {
    time timestamp;
    [float] left_input;
    [float] right_input;
    channel<obuf> reply;
}

struct obuf {
    [float] left_output;
    [float] right_output;
}

struct param {
    int param_id;
    float time;
    float value;
}

int ui_process(){
    m = ui_state()
    while(){
        select {
            case events.pop() {
                r = on_uievent(m, _)
                if(_.data.type == draw){
                    _.reply.push(r._paint_image)
                    m = r.m
                }
                else{
                    control.push_vec(r.control_params)
                    m = r.m
                }
            }
            case close {
                return nil
            }
        }
    }
}

int audio_stream_part(){
    m = audio_stream_ds(2, 44100)
    while(){
        select {
            case audiostream.pop() {
                m = process(m, _)
            }
```

```
                    case control.pop {
                        return nil
                    }
                    case close {
                        return nil
                    }
                }
            }
        }

        container = JSON
            [
                {
                    "doc": "you need to import pins.",
                    "label": "vsthost",
                    "type": "vsthost",
                    "def_pins" : [
                        [ "outpin", "request_param", "request_parameter()" ],
                        [ "inpin", "midi_in", "on_midi_input()" ],
                        [ "inpin", "set_param", "on_set_parameter()" ],
                        [ "inpin", "process", "process()" ]
                    ]
                },

                {
                    "doc": "A channel always have an input pin called *in* and an
output port called *out*",
                    "label": "events",
                    "type": "channel",
                    "element": "uievent",
                    "mode": "block"
                },
                {
                    "label": "control", "type": "channel", "element": "param", "mo
de": "block"
                },
                {
                    "label": "audiostream", "type": "channel", "element": "ibuf",
"mode": "block"
                },

                {
                    "type": "ui_part", "process": "ui_process"
                },

                {
                    "type": "audio_stream_part", "process": "ui_process"
                },
```

```
            {
                "type": "wires",
                "wires": [
                    [ "vsthost.midi", "control.in" ],
                    [ "vsthost.gui", "events.in" ],
                    [ "events.out", "ui_part.uievent" ],
                    [ "ui_part.control", "control.in" ],
                    [ "vsthost.set_param", "control.in" ],
                    [ "vsthost.process", "audiostream.in" ],
                    [ "control.out", "audio_stream_part.control" ],
                    [ "audiostream.out", "audio_stream_part.audio" ]
                ]
            }

        ]
```

NOTES AND IDEAS FOR FUTURE

IDEA: TASK DISPATCHER

This part lets you queue up tasks that have dependencies between them, and it will r ??? This only increases performance = part of optimization. Make this something you declare to expose potential for parallel task processing. Supports a stream of jobs, like a game engine does.

IDEA: SUPERVISORS

See Erlang. These start, monitor and fix containers and actors.

IDEA: DIFF AND MERGE

Diff and merge are important to container code to detect what changes needs to be performed in the world.

IDEA: All OS-services are implemented as clock:

you send them messages and they execute asynchronously? Nah, better to have blocking calls.

IDEA: The clock records a LOG of all generations of its state

...in a forever-growing vector of states. In practice, those older generations are not kept or just kept for a short time. RESULT: this is not done automatically. It can be implemented as a vector in clock's state.

IDEA: ADD SIMD FEATURES

IDEA: MAP() CAN RUN FUNCTION WITH SIMD INSTRUCTIONS

```
let pixel2 = map(pixels1, func (rgba p){ return rgba(p.red * 0.1, p.green * 0.2, p
.blue * 0.3, 1.0) })
```

Turns into

```
let count = size(pixels)
let batches = count / 4
var pos = 0
for(i in 0 .. batches){
    let vec factors = { pixels[pos].red, pixels[pos + 1].red, pixels[pos + 2].red,
 pixels[pos + 3].red }
    let vec reds = { pixels[pos].red, pixels[pos + 1].red, pixels[pos + 2].red, pi
xels[pos + 3].red }
    let vec greens = { pixels[pos].green, pixels[pos + 1].green, pixels[pos + 2].g
reen, pixels[pos + 3].green }
    let vec blues = { pixels[pos].blue, pixels[pos + 1].blue, pixels[pos + 2].blue
, pixels[pos + 3].blue }
    let vec alphas = { pixels[pos].alpha, pixels[pos + 1].alpha, pixels[pos + 2].a
lpha, pixels[pos + 3].alpha }

    let vec result = reds *
    pos = pos + 4
}
```

IDEA: TRANSFORMER MODES

1. sample-value: Make snapshot of current value. If no current value, returns none.
2. get-new-writes: Get a vector with every value that has been written (including double-writes) since last read of the transformer.
3. get-all-writes: Get a vector of EVERY value EVER written to the transformer. This includes double-writes. Doesn't reset the history.
4. block-until-value: Block execution until a value exists in the transformer. Pops the value. If value already exists: returns immediately.

READER modes:

- Default: reader blocks until new value is written be writer.
- Sample the latest value in the optocoupler, don't block.
- Sample queue of all values since last read.