Init System Proposal

For all POSIX complient systems

James Hobson and Michael Reim

May 29, 2022

Contents

1 Motivations **1**

2 Other Software in the Domain **2**

2.1 Primordial Unix init 2

2.1.1 Summary 2

2.1.2 Details 2

2.2 BSD rc 3

2.3 SysV init 3

2.4 BSD rc.d 3

2.5 OpenRC 3

2.6 Upstart 3

2.7 Systemd 3

3 Proposal for Utløse **3**

3.1 Architecture 4

3.1.1 Logging and alarms 5

3.1.2 Configuration 5

3.2 Interconnecting instances 5

3.3 Configuration Language 6

3.3.1 Service settings 7

3.3.2 Starting a service 7

3.3.3 Throttles 8

3.4 Service Management Tool 8

4 Configuration **9**

A Original init manpage **9**

Bibliography **11**

1 Motivations

Proposing new init systems and service management systems in 2022 is surprisingly still a taboo topic. This space has seen little innovation since the controversially wide adoption of `systemd` in linux and I think, rather understandably, no one wants another[1] init world war. But the war never really ended; while `systemd` gained control of vast amounts

[1] The systemd case may be what many people immediately think off as it kind of still rages on. It is however not the first fierce fight over init systems. In fact it feels a lot like the situation when the BSDs adopted `rc.d`. There have been fierce arguments including claims of this being ”not the BSD way” and threads to fork the old init system and continue on with it! For some reason or another, people tend to be extremely touchy when it comes to their `init` - and probably rightfully so!

of territory[2] and a kind of cease fire was put in place.

Despite having lost all the mainstream Linux distributions to systemd, enough people continue their resistance to make maintaining several alternative distributions feasible.[3] However there also exists a minority who still felt oppressed and jumped over to the BSDs to escape the creep of systemd. But this leaves the BSDs in a situation where, if they wanted to innovate their init system and service management, they would have to to try and carefully avoid emoting any PTSD in the refugees they got from Linux. These non-mainstream open source communities are likely too small to further fragment and remain viable[4]. Therefore an unspoken policy of init-system complacency has been adopted (i.e. it's regarded simple, well-known and for a lot use cases in fact *good enough*).

While some people argue that GNU/Linux is the only remaining Unix-like operating system that still matters, things are changing. The last couple of years have seen renewed interest in *BSD and an increase in newcomers. *FreeBSD* is attractive to many due to its superb integration of ZFS, its proven jails system and many other features. *OpenBSD* is relatively well known for its security-related innovations. *NetBSD* is a little less visible, but people are also finding their way to it. And even *Dragonfly BSD* has gone from a relatively unknown operating system to one that has started to out perform Linux in some benchmarks[5].

However there are a few things that arguably hinder a renaissance of the BSD. One thing that `systemd` has shown is that proper service management is key to speed and effective resource management. Therefore it may be time to take another look at init systems and service management so that the BSDs can have an up-to-date answer to the systemd problem and so that systemd has competition in it's domain. There are a couple of candidates to potentially fill that gap. It makes sense to take a closer look at them first.

2 Other Software in the Domain

2.1 Primordial Unix init

This init system, also known simply as *init* was there even in the earliest editions of Research UNIX but evolved over time.

2.1.1 Summary

The old UNIX init is extremely primitive. A benefit to this approach is that it's easy to understand as well as very stable. But the complete lack of service management has made it obsolete for a very long time. Some software, such as Apache HTTPd, came with management shell scripts[6] for service management back in the day, but for programs that didn't, your only option was to send the correct signal to the process via `kill`. For admins unfamiliar with this kind of management it's in fact easier to restart the whole machine after some service died than to figure out how to start it again!

[2] Some people claim that this happened not due to technical superiority. There is some truth to that and especially in the fight over *Debian* corporate actors (*Red Hat* for `systemd` and *Canonical* in favor of `upstart`) were not hard to spot. But it is also true that `systemd` *does* provide a lot of value over `SysV init`.

[3] There are those like *Slackware* which has used its BSD-inspired init system before `systemd` entered the stage and continues to do so. Other such examples are *Gentoo* (by default) and *Alpine Linux* which uses `openRC` and *Void Linux* that adopted `runit`. There are however even new *systemd-free* distributions which consider this an important feature. Notable examples are the Debian fork *Devuan* and the Arch Linux fork *Artix*.

[4] *FreeBSD* as the by far largest player has seen several failed attempts of getting `openRC` into the base system. *TrueOS*, a friendly fork that received corporate sponsorship has made completed the switch – but it died not too long afterwards. *GhostBSD*, a desktop-focused distribution of FreeBSD picked it up and maintained it for years but eventually even migrated back mainly because of the maintenance burden.

[5] Michael Larabel. *DragonFlyBSD 6.0 Is Performing Very Well Against Ubuntu Linux, FreeBSD 13.0*. May 2021. URL: `https://www.phoronix.com/scan.php?page=article&item=corei9-freebsd13-dfly6&num=4`.

[6] *Apache HTTP server Version 2.0 - Stopping and Restarting*. URL: `http://httpd.apache.org/docs/2.0/stopping.html`.

2.1.2 Details

In the original First Edition of Research UNIX, init was a self-contained program that was responsible for all the tasks required to bring up the system. Since it is of high historical value, we replicate the information from the manual in §A.

2.2 BSD rc

BSD innovated by adding `rc.local`. This separated system init and user specified init, thus removing the anxiety sysadmins would face each time they needed to update the system. Apart from that change, it was pretty much the same as old Unix init[7] and so inherited the lack of features with respect to service management.

2.3 SysV init

SysV init added the concept of runlevels[8] and basic service management[9]. A run level is a defined state that includes a set of services which have to be running (or have to not be running). Changing run level is a simple as stopping the services that require stopping and starting the ones that are not running. The issue with SysV-style init is it's implementation and how basic the service management is. It over uses symlinks in the file system to determine order, it uses PID files to keep track of processes. On top of this, the service management is quite primitive.

2.4 BSD rc.d

It provides the main benefit of breaking up formerly monolithic rc into several init scripts. Unlike SysV init, it is configured centrally in a single (from the user perspective) file: rc.conf. It also features the `rcorder(8)` tool and can generally figure out itself what the right order to start services in is. Thus it avoids the most serious problems with SysV init. But it's somewhat limited; there is no service supervision, service status is not actually reliable and it's not helpful to get a full system overview. Parallelism is an afterthought and a somewhat recent feature addition.

2.5 OpenRC

Good: Introduces some convenience functions like rc-status. Was conceived with allowing for parallelism. It can work together with other programs to allow for service supervision. Bad: Picked up the SysV folly of service enabling via symlinks. It is still unreliable with service status. By itself it's not able to supervise services.

2.6 Upstart

Good: Compatibility with SysV init scripts. Asynchronous, event-driven nature. Service supervision. Bad: Hm! It's been a while...I don't remember off the top of my head what I disliked. Would need to look into it again.

2.7 Systemd

Good: Transition from init scripts to unit files. Service supervision. Bad: Braindead feature creep (for a PID 1 process!!). Very strange random defects (= unreliable). There's much more both for good and bad, but these are the main 4 points about systemd for me.

[7] *init.* June 2021. URL: `https://en.wikipedia.org/wiki/Init`, Research Unix-style/BSD-style.

[8] *init.* June 2021. URL: `https://en.wikipedia.org/wiki/Init`, SysV-Style.

[9] *service(8).*

3 Proposal for Utløse

The aims of this project is to write a research init system called Utløse. It will provide two things: A way to start services and a way to monitor and manage them. These components have the following aims

- To abstract away order and state from the system
- To start the minimum possible number of services
- To be as parallel as possible
- To assist the user at every complexity level
- To allow for maximum customisability
- To be fail-safe. If PID 1 crashes, *hopefully*, it will not require a reboot.
- To be POSIX compliant and as flexible with licensing as to not lock ourselves to a particular operating system
- To intergrate well into package managers
- To allow for central monitoring and control for distributed systems

Unix has a long history of using programming language theory to solve problems. As soon as lex and yacc, were available, tools such as awk were written. In the case of awk, a language for text processing and report generation was written. The language abstracts away all that gets in the way of it's aims so that it can do what it does well. This is in contrast to most of the available init systems, which tend to be build upon pre-existing technologies (such as shell and the file system) or use extremely generic configuration formats (systemd uses the ini format). As a result, very little complexity is abstracted away. We propose a new language designed for service management.

The first idea for this language is that dependencies are captured in the syntax. The next idea is that the language is lazily evaluated. This means that everything is started as late as possible (if started at all). Automatic parallelism and lazy service starting should mean lighting fast boot times and minimum resource usage. The language will be garbage collected. But what is the garbage? The garbage is services that are currently not in use. Will maintain minimum resource usage. Another idea is finite resource limits. Consider a large number of services that heavily use the disk or network during initialisation. We could tell the scheduler that these services take from a finite resource. The scheduler will make sure that only a set number of these services run at once. This should mean that hardware bottlenecks in the system do not end up slowing down boot.

The language will also allow for other interpreters to be used in the definition of services. If you have a bunch of complex procedures that need to be run to calculate how a service must start, then you can use haskell, lua python (etc). Or you can keep it simple and let it default to shell.

We may explore disowning children processes. This has some limitations, but possibly would allow the daemon to fail without the system being rendered unusable.

3.1 Architecture

There will be two executables:

- `utlosed`
- `utloseadm`

Where `utlosed` will be the daemon that controls the starting and maintaining of services, and `utloseadm` which is the frontend to be used by both users and other applications. The two applications will be connected via ZeroMQ[10]

[10] *ZeroMQ*. URL: `https://zeromq.org`.

sockets. There will be request and reply sockets for control, but also publisher and subscribe sockets. These will be used for distributed logging and alarms. ZeroMQ is a good choice because of curveZMQ[11]. Encrypted sockets (to allow for distributed control) can be achieved with only minor changes to the code base.

The messaging protocol is not decided yet, but it will be open. This will allow for third parties to easily write other frontends, or log collectors.

3.1.1 Logging and alarms

All output of the services will be redirected back to `utlosed` which will both write to a rotating log (configurable) and broadcast. The channel will be configurable, but the default will be: `logs/hostname/service`. This means that any logging client can filter for logs, logs from a specific host, and logs from a specific service on a specific host.

Logs written to `stderr` (and crash reports detected by `utlosed`) will also be reported on `alarms/hostname/service` allowing for a service to listen specifically for things that are going wrong.

All logs currently held on disk may be requested via the request response socket interface. This allows remote clients to both respond to live events happening (via subscription), without having to be running 100% of the time to get a full history.

3.1.2 Configuration

Configuration is split into three parts:

1. Configuration of `utlosed`
2. Configuration of `utloseadm`
3. Configuration of the services.

Configuration of the first two will be found in `/etc/utlose`, but service configuration can be split accross locations. This is to appease some package managers, which like to place everything in `/usr`

By default, `utloseadm` will assume that it is monitoring a local instance of utlose. This can be changed by adding hosts to `/etc/utlose/utloseadm.toml`.

The config for `utlosed` will be stored in `/etc/utlose/utlosed.toml`, and will be used to turn on or off features, change time outs and specify where the service files are located. This will also be where the default *rule* is specified.

There are a few approaches to service management config that could be appropriate. I imagine that a few will be supported and the one used will be configured in `/etc/utlose/utlosed.toml`

- Subvolume based upgrades: File Systems such as ZFS and BTRFS support snapshots of subvolumes. In this option, a snapshot is created every time utløse successfully loads a new complete service configuration. If it fails, it goes through the snapshots until it finds one that works.
- Different locations: In this option, the package manager submits changes to Utløse which are accepted or rejected. The mechanism here would be that the package manager puts a service file in some pre-agreed location and calls a subcommand of `utloseadm` to query the new service. The query can be simple, such as "Is this well formed?" But more complicated queries can also be supported. If all the queries pass, the new service files are included. If not, they are rejected and the package manager warned.
- Nothing special: Keep it simple stupid! One location for service files.

[11] *Curve ZMQ*. URL: `http://curvezmq.org`.

3.2 Interconnecting instances

As outlined above, logs can be broadcast to other instances, but this is not where the interconnecting ends. Utløse instances can be chained and control commands can be send down the chain. The best way to explain this is by example:

Consider a container (such as a jail), on a virtual machine in a datacenter. On this container, we run NGINX. The Utløse instance on the container has a service named NGINX. But we have connected this instance with the Utløse instance in the VM.

The VM starts this container via Utløse and calls the container “frontend”. But because the instances are connected, on the VM, we can also see the service `frontend/nginx`. If allowed, from the VM, we can issue commands to the `utlosed` daemon using the `utloseadm` on the vm, such as `utloseadm status frontend/nginx`.

The VM is on a node in a datacenter. It’s name is “FreeBSD_12”. The node’s instance is connected to the instance on the VM and so on the VM we can see the NGINX service as `FreeBSD_12/frontend/nginx`. We also have a computer in charge of collecting logs. This computer can query the node in much the same way.

Of course, permission and security are of paramount importance. For this reason, the connection between the nodes are end to end encrypted and what a node up the hierarchy can do is controlled by the `utlosed.toml` file.

3.3 Configuration Language

The configuration language is inspired by the shake build system EDSL. Services are defined within targets, both of which can be dependencies of services. Target definitions can be split allowing for configuration to be split accross files and are defined with capital letters at the begining of their names. Services are defined within targets and cannot be split. The general syntax is:

```
Target => -- Name of target
  wants [service1, service3] -- what needs starting in target.

  service1 => -- Service name
    needs [someTarget, someService] -- dependencies
    -- Service definition follows

  service2 =>
    ...

  ...
```

both `wants` and `needs` are optional. But you will probably want to use `needs` unless you are writing a service to be started right after utløse starts!

So how can a package possibly add a service? It seems like it requires to *edit* a file instead of adding a standalone one! Do not fear. Targets can be merged accross files for example:

```
-- /etc/utlose/services.utl.d/10_bla.utl
  Target =>
    wants [foo]

    foo =>
      ...

-- /etc/utlose/services.utl.d/20_other.utl
```

```
  Target =>
    wants [bar]

    bar =>
      ...
```

Is equivalent to

```
Target =>
  wants [foo, bar]

  foo => ...
  bar => ...
```

Files are included in lexicographical order, so if there is a conflict, the second file takes precedence. `/etc/utlose/services.utl` is loaded after all of the ones in the `services.utl.d` folder and so takes precedence. This is for user defined services.

3.3.1 Service settings

The following things are configurable by a service:

- Run user, the user that the service runs as
- Service type, default is simple, other option is scripted
- Command, what to run!
- Socket, this is for the lazy starting of services network serivices. A port is defined and a unix domain socket as the proxy
- Sockets, same as socket but accepts a list of socket mappings as opposed to 1.
- gcTime, if the service is started lazily, kill it if no activity on the socket after the given time
- version, takes either a path to a file containing the version, or a version number. When a user updates a group of services and wants to restart 1 of them, Utløse can be told about version dependencies. This allows utløse to restart all of the required services.
- timeout, how long to wait for service before giving up. (overrides default in `utlosed.toml`)
- Restart, how long to wait before restarting. If not present, don't bother.

```
SomeTarget =>
  someService =>
    needs [Network, foo >= 0.3.2]
    version = 1.2.1
    user    = "root"
    type    = SIMPLE
    socket  = 0.0.0.0:8080 -> /root/fun.sock
    gcTime  = 1 days
    restart = 10 secs
    command = /usr/bin/funservice --listen-on /root/fun.sock
```

3.3.2 Starting a service

A simple service launches a program and seems the job done. Often, you might want to wait for something to happen before you deem the job done. How this information is passed to utløse is up to the programmer, as this is done through scripts:

```
SomeTarget =>
  someService =>
    needs [Network, foo >= 0.4.0]
    version = 1.2.2
    user    = "root"
    type    = SCRIPTED
    socket  = 0.0.0.0:8080 -> /root/fun.sock
    gcTime  = 1 days
    restart = 10 secs
    command =
     [/bin/bash |
        /usr/bin/funservice --listen-on /root/fun.sock &
        while [ ! -f /etc/fun/started ]; do
          sleep 1
        done

        fg
     |]
```

You can use any interpreter is appropriate. This should mean that no program needs to depend upon utløse to be integrated well. An aim of this project is portability and this means software written for an Utløse system is in no way bound to it.

3.3.3 Throttles

We may want to limit the number of web services that start simultaneously on boot. Maybe this is because we have limited bandwidth and it would slow down boot. We might also have disk heavy services that we want to limit. For this we use finite resources:

```
net  <- resource 10 -- 10 network services max at a time
disk <- resource 4

Product =>
  frontend net =>
    ...
  database net disk =>
    ...
```

Throttles may want to be inherited across interconnected instances. This can be configured in the `/etc/utlose/utlosed.toml` file.

3.4 Service Management Tool

Management is done via `utloseadm`. It has the following subcommands.

- `start` - start service

- `stop` - stop service
- `restart` - restart service
- `status` - query status
- `logs` - logs for services
- `lint` - generate errors and warnings
- `snap` - save current service configuration for auto-rollback
- `revert` - rollback
- `enable` - enable a service at startup
- `disable` - disable to the autostart of a service
- `patch` - add a file to config. Lint it first. If successful, snap and then add it.
- `query` - query properties about the service system e.g. timings, dependencies and more.

Most subcommands will have a `assert` and `check` option which can be provided with a precondition or post condition. It is not yet decided what the language will be for these conditions, but a pseudo example would be `utloseadm start --assert 'wicked down' NetworkManager`. This will check that the system hasn't got the wicked service running, before attempting to start NetworkManager. I imagine that this feature would only help packaging macro authors.

4 Configuration

Example:

```
instance = "Name instance" # defaults to hostname
default_timeout = 3
service_files = ["/etc/utlose/services.utl"] # .utl.d folders are assumed. Order matters!
log_folder = "/var/log/utlose"
public_ip = "0.0.0.0" # If not present, doesn't bind to any port
public_port = 9000

# Permissions for interconnected instance parents
[permissions]
status = true
control = false
logs = true
# How many levels up can see this instance. 0 for all
visibility = 0

[interconnects]
key = "XXXXXXXXXXXXXXX" # private key
resource_limits = ["net", "disk"] # Resources to share with children

# settings for connecting to children
[[child]]
  key = "XXXXXXXXXXXXXXX" # public key
  ip  = "0.0.0.0"

[[child]]
  key = "XXXXXXXXXXXXXXX" # public key
  ip  = "0.0.0.0"
```

A Original init manpage

Keep in mind that the sections weren't what we're used to today and those sections were denoted in Roman numbers. Two other curious things are that the init binary used to be located in /etc and that originally there was no *group* bit for file permissions.

Here is the manual for init as of the First Edition[12]:

```
11/3/71                                     /ETC/INIT (VII)

NAME          init -- process initialization

SYNOPSIS      --

DESCRIPTION   init is invoked inside UNIX as the last step in
              the boot procedure. It first carries out several
              housekeeping duties: it must change the modes of
              the tape files and the RK disk file to 17, be-
              cause if the system crashed while a tap or rk
              command was in progress, these files would be
              inaccessible; it also truncates the file
              /tmp/utmp, which contains a list of Unix users,
              again as a recovery measure in case of a crash.
              Directory usr is assigned via sys mount as
              resident on the RK disk.

              init then forks several times so as to create one
              process for each typewriter channel on which a
              user may log in. Each process changes the mode
              of its typewriter to 15 (read/write owner,
              write-only non-owner; this guards against random
              users stealing input) and the owner to the
              super-user. Then the typewriter is opened for
              reading and writing. Since these opens are for
              the first files open in the process, they receive
              the file descriptors 0 and 1, the standard input
              and output file descriptors. It is likely that
              no one is dialled in when the read open takes
              place; therefore the process waits until someone
              calls. At this point, init types its "login:"
              message and reads the response, which is looked
              up in the password file. The password file con-
              tains each user's name, password, numerical user
              ID, default working directory, and default shell.
              If the lookup is successful and the user can sup-
              ply his password, the owner of the typewriter is
              changed to the appropriate user ID. An entry is
              made in /tmp/utmp for this user to maintain an
              up-to-date list of users. Then the user ID of
              the process is changed appropriately, the current
              directpry is set, and the appropriate program to
              be used as the Shell is executed.

              At some point the process will terminate, either
              because the login was successful but the user has
```

[12] Taken from here: https://www.tuhs.org/Archive/Distributions/Research/Dennis_v1/man71.pdf

now logged out, or because the login was unsuccessful. The parent routine of all the children of *init* has meanwhile been waiting for such an event. When return takes place from the *sys wait*, *init* simply forks again, and the child process again awaits a user.

There is a fine point involved in reading the login message. UNIX is presently set up to handle automatically two types of terminals: 150 baud, full duplex terminals with the line-feed function (typically, the Model 37 Teletype terminal), and 300 baud, full duplex terminals with only the line-space function (typically the GE TermiNet terminal). The latter type identifies itself by sending a line-break (long space) signal at login time. Therefore, if a null character is received during reading of the login line, the typewriter mode is set to accommodate this terminal and the "login:" message is typed again (because it was garbled the first time).

Init, upon first entry, checks the switches for 73700. If this combination is set, *init* will open /dev/tty as standard input and output and directly execute /bin/sh. In this manner, UNIX can be brought up with a minimum of hardware and software.

FILES	/tmp/utmp, /dev/tty0 ... /dev/ttyn
SEE ALSO	sh
DIAGNOSTICS	"No directory", "No shell". There are also some halts if basic I/O files cannot be found in /dev.
BUGS	--
OWNER	ken, dmr

Bibliography

[1] Michael Larabel. *DragonFlyBSD 6.0 Is Performing Very Well Against Ubuntu Linux, FreeBSD 13.0.* May 2021. URL: https://www.phoronix.com/scan.php?page=article&item=corei9-freebsd13-dfly6&num=4.

[2] D. M. Ritchie K. Thompson. *Unix Programmer's Manual.* Second. Bell Telephone Labpratories. June 1972.

[3] *Apache HTTP server Version 2.0 - Stopping and Restarting.* URL: http://httpd.apache.org/docs/2.0/stopping.html.

[4] *init.* June 2021. URL: https://en.wikipedia.org/wiki/Init.

[5] *service(8).*

[6] *ZeroMQ.* URL: https://zeromq.org.

[7] *Curve ZMQ.* URL: http://curvezmq.org.