# Waltham evaluation

*by Gunnar Andersson. Work in progress. Details may be changed when misconceptions are corrected.*
*All of it to be discussed with GSHA team, experts, possibly Waltham author, etc.*

This is based on browsing the code and documentation.

## Summary:

Waltham code is described simply as: "**IPC library**" *****.

The application protocol itself is decoupled from the implementation of data transfer. I don't mean just you can use different wires (everyone does that), but that significant parts of the lower-level protocol that is Waltham is independent of what IPC/RPC interface is actually transferred.

The Waltham code implementation is thus

- A low-level connection-oriented client-server protocol that carries an application protocol on top of it - implementation provided as a library.
- *+* separate supporting code for defining and implementing that application protocol.
- Finally, with this functionality, the *project that* is Waltham, is tied into the particular use case of transferring a Wayland-compatible protocol, (plus optional extensions) as the application protocol.

The library that handles the transport is interestingly oblivious to the application protocol. As a concrete consequence, the waltham library should not need to be recompiled because the application protocol changes.

Waltham is thus independently extensible. The interfaces provided by the application protocol are described in XML files. Waltham provides per default, XML files to mirror appropriate Wayland protocol functions. The adopter is suggested to extend it with other functions as needed and that doing so mirrors the Wayland plugin mechanism.

Waltham is set up to do communication over a communication link and in Linux (for example) the "file descriptor" abstraction and an identical read/write API covers all of those cases with the same APIs. Waltham could thus be applied locally, or remotely although the driving use case is of course extending Wayland across a network of some kind.

***** Side note:*

*Inter-process communication is often considered to be between processes on the same operating system instance, whereas communication over a network to a different system uses other names like Remote Procedure Call (RPC). In this case (and probably in most these days), the distinction becomes less and less important. The highly optimized data encoding and handling in Waltham seems to be efficient in both local or remote context and the generic API concepts lend themselves well to flexibility. At the lowest level, the implementation is a **readv()/writev()** call (read/write functions operating on an array of buffers at once). These generic calls always operate on a file descriptor, which in UNIX/Linux could just as well represent a local pipe /socket, a network socket and other transports. The Waltham data transfer reduces to transferring all data "at once", at whatever efficiency the underlying transport has. Presumably a shared-memory implementation might even be added with small changes?*

*However, that is getting a bit far off since a Wayland-speaking compositor itself is presumably available locally, so the case of "remoting" the Wayland API makes little sense locally. The reason I bring this up this discussion at all is that it's a little bit unclear to me what the primary project intention is - Wayland only, or other IPC/RPC? Network/remote only, or also local IPC? (see Questions/discussion section)*

## Detailed Functionality

A majority of the implementation deals with:

- APIs that support clients and servers to set up, and track, their connections.
- Supporting servers to keep track of their list of multiple clients
- Some associated development support, tracing/logging the connection state etc.
- Low-level protocol details, such as invalidating the connection if mismatched data is received, such as an unknown function (server/client using different protocol definition)
- Service announcement? - server letting client know what operations are supported *(⚠️ I need a bit more understanding of that part)*

An *ad-hoc* code-generator written in single file python program takes the XML files and generates functions for (de)serializing the interface parameters, assigning unique IDs to each new function, etc. Once generated, the application protocol is static and no extension possible during runtime. The client and server implementation must use the same code generation input since things like function IDs are assigned simple integers that must match.

As stated, the library does not deal with any application protocol details but only knows the low-level Waltham protocol details - the understanding of a connection, and transfer of single messages (e.g. one remote-procedure call - although note that Waltham is asynchronous).

The Waltham library is low-level and direct transfer of structured data. Function name/ID is followed by a list of data items that can be simple integers or arrays or structured data, (which is just flattened to a byte array too). At the lowest basic level, Waltham primarily knows that it shall transfer a function ID, and buffer(s) of bytes of a particular size. Setup and teardown of the client-server connection is also handled by the Waltham library APIs.

Serialization and deserialization is done by functions provided by the code-generated files that are specific to the protocol. The generated code in turn uses macros that implement the serializing/deserializing. These macro implementation is provided by waltham source code (header file) but need not be part of the library as previously explained since the library only needs to be able to transfer the flattened data).

## Documentation and usage

Documentation is short but highly technical and to the point. There is little superfluous information about the philosophy or context. The documentation feels as written for developers with extensive system programming/library development experience.

Integrating Waltham requires UNIX/Posix system programming knowledge. Basic kernel API characteristics such as file descriptors, select/poll, timers, etc. is the language exposed to the users of the Waltham API. The consequence is a lot of flexibility, but with some technical hurdle to get started. For developers who are system library writers, everything should be familiar, whereas for others the code-style and usage is likely advanced and challenging.

By assuming an interface based on fundamental features like file descriptors, etc, Waltham does not assume much from the environment it is used. This should aid flexibility and portability, but I would say primarily to somewhat similar systems (UNIX/Posix). Operating-system concepts are somewhat exposed and there is no major abstraction layer included to place adaptions to, for example a non-POSIX operating system. That is clearly a design choice and is in line with Waltham's simple, direct and efficient implementation.

## Waltham plugins

Waltham library should be used from the concrete Waltham-plugins. There are two major type of plugins:

- waltham receiver (or waltham server)
- waltham transmitter (waltham client)

One simple example of the responsibilities of the plugins can be as following waltham transmitter will have access to the graphical content and will pass to the waltham receiver. Waltham receiver will display the content and will provide the input event back to the transmitter if corresponding content is touched.

Plugins have to implement the same waltham protocol to be compatible to each other.

The protocol can carry different type of data:

- semantic information of the scene changes (new application, orientation change of the display)
- input events
- the graphical surface data itself (should be avoided in the virtualization environment: better way to the share graphical content is expected since access to common memory is possible)

24.05.2018 current implementations of the plugins:

- waltham transmitter plugin for weston 2.0(still in review by AGL https://gerrit.automotivelinux.org/gerrit/#/c/13767/)
- waltham receiver (TODO)

## Discussion

(Questions, Criticism, Concerns... open for discussion)

### General observations

1) Surprisingly to me, a non-trivial part of the implementation deals with implementing basic datatype and algoritms such as linked lists and other linked data structures.

Thoughts:

- The implementation looks highly professional, but is as always a cause for concern, until proven to be 100% debugged.
- For C programs, this is common practice, but not necessarily a good one. (The lack of a standardized library such as C++ provides is part of the problem.)
- Possibly the open-source developers deal with this issue by copying of such implementations from other projects(?). Without that, it seems a large unnecessary effort to reimplement basic algorithms and prove its correctness.
- The end result however reduces dependencies and enables portability. Waltham would be way less portable had it depended on glib, or something like that.

2) Both API design and the given client/server examples are clearly tailored to relatively advanced Linux system programmers that are immediately comfortable with the APIs of timers, epoll, the meaning of the "watch" structure, and similar details. The concepts, naming, and structures may be hard for general programmers to get used to. I think this is is likely something that has to do with experience, but I would still say it's not for the faint of heart.

3) The implementation of the IPC/RPC is laudably generic (interoperability concerns below) and judging from source-code analysis seemingly **very** bandwidth and CPU efficient, as well as extensible. However, the stated project intention is also to do remoting of Wayland. Could it be clarified to what extent this project should remain Wayland-specific and to what extent it's intended to be "An IPC mechanism" (as the project title says).

- - Are some parts of the program only dealing with Wayland specifics? Despite the Waltham protocol being transparent to what is being sent on it, there are other significant parts of the source code speaking about things like "displays"? These are not in the server/client example parts but rather part of the Waltham code itself? What is the significance of that?
- The Wayland protocol is included as the default XML files.
- Is the program mixing generic IPC parts and Wayland-intended parts unnecessarily, or is this the intention?
- Is it intended and appropriate to use this implementation as a generic IPC/RPC implementation, totally independent from the Wayland remote case? What parts of the code would better be extracted and isolated from Wayland specifics, were that the case?
- How do we ultimately agree on recommendation for usage. In other words - in a typical system, what IPC/RPC features should go in as a Waltham protocol extention, and what features are better placed elsewhere? Is the answer "anything graphics related"? or something else?

4) Would the author comment on not basing this transport on existing IPC/RPC systems with similar efficient encodings (Protobufs, Flatbuffers, Cap'n proto...). Wouldn't there by good synergy effects in reusing existing technologies here, considering available tooling, ecosystem, language bindings, etc.? (This ties back to where to put the non-graphics IPC/RPC - shall that use Waltham or some other methods?)

5) After deciding on Wayland-specifics vs generic IPC, I already now think a clearer separation of parts might be possible.

- The generated code depends on certain header files (e.g. marshaller.h) but presumably the library does not need that file.
    - (why does waltham-connection.c include marshaller.h – does it really use it?)
- In other words, source code for the library (stated to be compilable once with little connection its usage) seems mixed with source code for the application-protocol (which is independent of library code?), as well as support for clients, servers, etc.
    - Although, tests/ directory correctly separates example client/server code from the rest.
- The header file(s) that expose the external API could be placed separate from internal (and/or generated) header files?
- Is there better separation possible here, or is it just a misunderstanding?

## Concerns about portability and cross-domain usage

1) There is no explicit handling of network byte-order. Presumably nowadays, 95% (*totally made up)* of systems are little-endian, so one could assume that both sending and receiving side is? Is that the current consensus in the industry?

- Adding to the previous, since Waltham appears to just serialize structured data by reading from a pointer to the struct in memory - even with identical byte order, how can we address concerns about data being encoded differently in memory if compiled with different compilers and on different systems?

According to the author, is it:

- - known to be **safe** (explain how it works, please :)
- - known to be **unsafe**, and this is a deliberate design choice
  (Can we then get the requirements for system and compiler similarity well defined?)
- - or known to be safe in certain scenarios
  (again, define boundary conditions)

2) I would appreciate an explanation of the **padding** feature. When is data padded and how (and does that address the struct memory layout issue?). Fully understanding this from the code proved challenging, so it'd be better to hear it from the author or another expert.