Distributed System Design in an Unconventional Medium
Greg Sinclair

Abstract

A distributed real-time control system is implemented in a non-standard medium and then optimized to the parameters of that medium. The resulting system will be compared to a standard distributed system, exploring the impact of medium-specific concerns on system design both in terms of specific implementation details and broader design concepts.

1. Motivation

While the origin of this project is overtly gimmicky, it employs real design principles and presents insights into how systems develop based on their medium. Particularly, this project shows that a classical distributed control system, when built using a contrived abstraction of digital circuit elements rather than the elements available in the real world, presents very different concerns and ultimately optimizes toward a solution that may be unexpected. To clarify, this paper discusses a distributed control system constructed within the simulation game “Factorio” by Wube Software LTD, intended to efficiently regulate an in-game train network. This paper will abstract the digital circuit implementation and the train pathing algorithm in Factorio so that readers unfamiliar with the game need simply keep in mind the basic elements available in this project. Any other details pertaining to Factorio are considered beyond the scope of this paper.

As to why Factorio was actually chosen for this project, the simple answer is that I happened to be experimenting with control systems for a Factorio train network when it came time to do a term project for an undergraduate engineering course at the University of Manitoba on Distributed Systems. The course material was inspiration to formalize the project into a proper, scalable distributed system, and the professor was amused enough by the concept to allow it as a term project. After the term project, featuring a single client reacting to “mocked” server commands, was well-received, I gradually completed the project both for my own use and as a fun portfolio item.

This paper will outline the problem requirements and the medium-specific tools available, then develop the system at a high and low level based on emergent requirements. Finally, the resulting system will be contrasted with comparable real-world systems and the realizability of this specifically optimized architecture will be discussed.

This project was developed in the version 0.17 branch of Factorio in 2020, and it is worth noting that the version 1.1 branch released in 2021 changes some of the project requirements. Since features implemented in 1.1 may potentially trivialize the project, this discussion refers to the 0.17 branch unless otherwise stated.

2. Problem Statement

The service provided by this system can be summarized as a controller for a logistics train network. In train-based Factorio worlds, the fundamental goal of the train network is to load

resources from facilities that produce a resource and transport them to facilities that consume the resource. If this is done optimally, both facilities will run at full capacity (the former never exceeds local storage and bottlenecks, the latter never depletes local resources and halts production). This problem can be generalized to M producers and N consumers of the same resource, ideally selected to match production and consumption of resources, and the train network must reduce the total downtime of all facilities to ensure maximum productivity of the factory as a whole. Since there is more than one resource type to handle, the system must also be generalized to K unique resources, but for practical reasons the solution to this is to create an independent system for each resource and superimpose K instances to cover all resources. Thus, a single-resource system will be discussed, under the assumption that for a reasonable value of K the single-resource solution naturally scales into a K-resource solution.

It must be noted that this system is constructed atop Factorio's inbuilt train pathing system. Without getting into the details of this rudimentary system, let alone the techniques used to generalize a hard-coded train schedule into an adaptive and scalable network, the trains as a system element can be described by the problem requirements they present and the methods by which they can interface with system elements. These details are informative in deriving a problem statement, but it will be shown later that a solution can be abstracted in such a way as to not be explicitly concerned about trains and their algorithms. For this reason, the details of train behaviour can be found in Appendix 1 and we move directly to the solution parameters they imply:

1) All stations will be disabled unless the system is explicitly sending them a train.
    i) Any station which already contains a train is disabled.
2) Trains are kept idle until dispatched to a station.
    i) In order to decouple the ratio of trains and stations, the location at which these trains idle will not be a station.
    ii) The trains will idle at a designated area called a trainyard, which is a system client akin to the stations.
    iii) While idling at a trainyard, trains may either be empty or full.
3) The system must manage all stations, dispatching trains based on the resource count at a station.
    i) A station producing iron must be dispatched an empty iron train upon local reserves rising above a certain value.
    ii) A station consuming iron must be dispatched a full iron train upon local reserves falling below a certain value.
    iii) Once the transfer is complete, said train will immediately be dispatched to a trainyard.

A realization of this system can be seen in Figure 1.

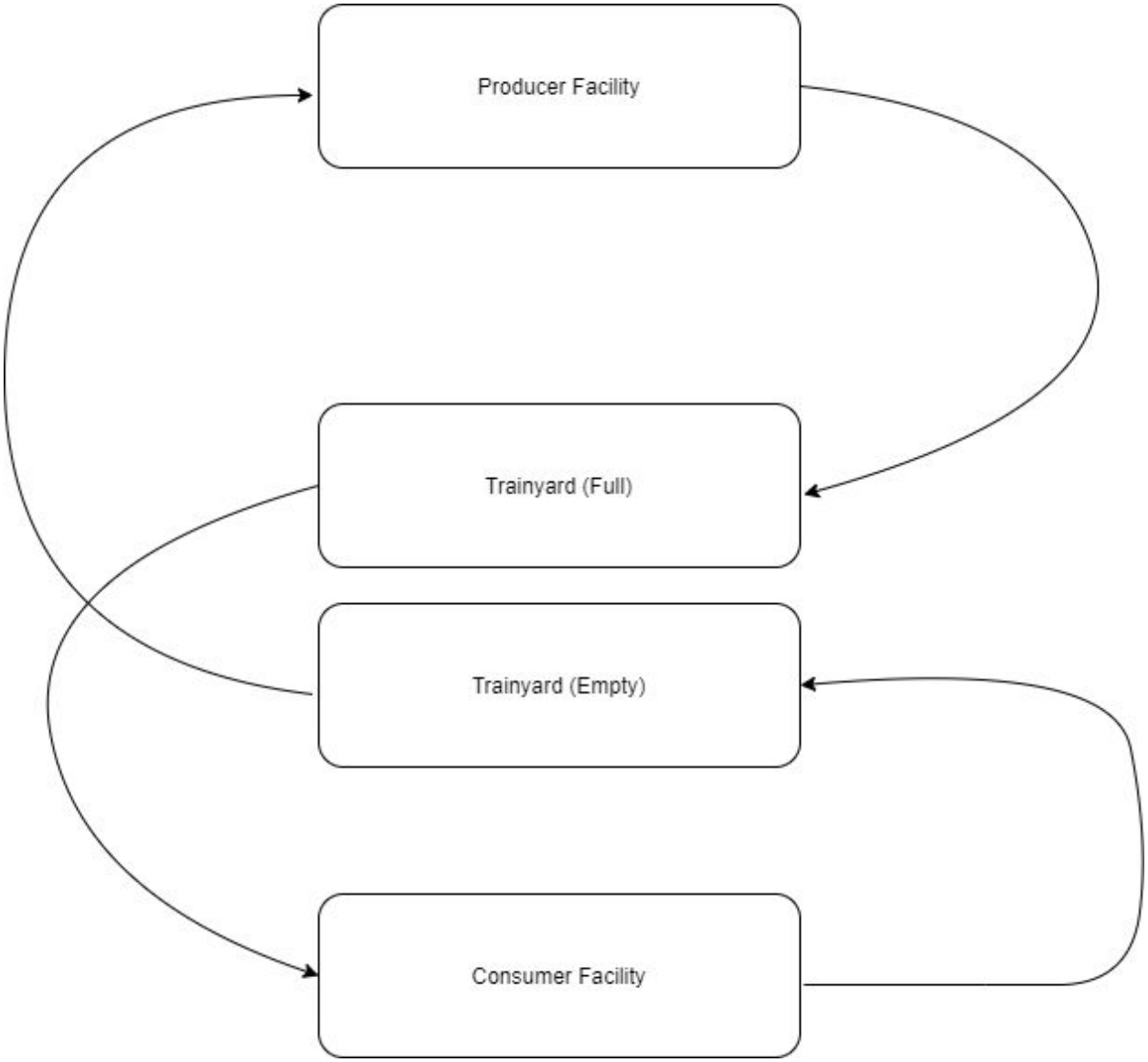

Figure 1: Train pathing sequence.

In practice, the trainyards for empty and full trains are overlaid. This setup can easily be generalized to M facilities and N trainyards. The associated train schedule can be seen in Figure 2. The variable F is a local release condition indicating that the train has permission to leave the current station.

Figure 2: Factorio implementation of the train route

The task, then, is to implement a distributed system that tracks the status of each station. When a train is needed at a station, that station is enabled and a train from an appropriate trainyard is released such that it will route to that station. In this way, the issue of trains incorrectly routing to the same station and queueing is eliminated. While scenarios in which two stations enable at the same time and two trains both route to the same station can be imagined, this will resolve when the first train arrives, the station disables, and the second train automatically reroutes to the intended destination. Given the speed of Factorio trains compared to the frequency at which stations actually require resources, such instances should be rare and also settle with little cost to productivity. As an additional safeguard, the system will periodically suspend operation to perform self-configuration, which gives hypothetical endlessly roaming trains adequate time to settle.

Before such a system can be implemented, the tools available must be described.

3. Circuit Mechanics

Factorio contains a small number of circuit elements, from which larger systems can be synthesized. The particulars of how these elements run in the game engine presents a number of unusual opportunities and challenges compared to designing with realistic electronics. In order to explain these differences, it is necessary to understand the circuit components Factorio offers.

The wire is the backbone of the circuit network. Considered in terms of digital circuits, it is a communication channel that is perfectly accurate and communicates instantaneously over any distance. This is clearly a fantastic tool compared to any physically realizable communications channel, and can be exploited thoroughly toward scalability of this system. There are two wires, the Red Wire and Green Wire, which are useful for isolating variables or processes. Each wire supports 48 user-defined variables, each 32 bit integers with range -2,147,483,648 to 2,147,483,647. Spreading a single global wire allows for easy communication between all Client and Server entities, but there are downsides that must be considered. The variables on the wire are not variables in the programming sense, but rather sums of current inputs to the wire. That is to say, if four entities push to the wire that “A=1”, the variable A on the wire will have value 4, and only so long as those entities continue to push their messages. If an entity sees that the wire holds “B=1” and desires to set the variable to 0, it must push “B=-1” and then maintain this equilibrium.

Clearly, the simplest way to avoid chaos on the global wire is to only allow a single entity to place messages on the wire at a time. This suggests a single-master bus architecture that uses polling. Though the fact that memory cells can be synthesized with circuit elements suggests that higher-level architectures are possible, the delays induced by complex structures render memory cells useless for real-time interfacing.

The remaining circuit elements are the logical components which push values to the wire. There are two such elements, the Arithmetic Combinator and the Logical Comparator. These can, however, be abstracted into the operations they enable without getting bogged down in the specifics of Factorio’s implementation. These operations include standard integer arithmetic such as addition, division, and modulus division. Operations can be performed conditionally, where the condition is checked by comparing an input to a specified value, or another input. Combining variables, moving values from one variable to another, and similar can be accomplished as well. Clearly, complex algorithms can be realized from sufficiently many circuit elements. Timing considerations, however, render these algorithms extremely slow if architectures are not specifically optimized.

4. Propagation Time

Being a simulation but not one intended for electrical engineering, Factorio has a fundamental time constant that is considerably longer than in genuine electronics. Specifically, each circuit element one full frame render to act on changes at the input. With Factorio running at 30 frames per second, three elements performing some operation in series will take a full tenth of a second to complete. Since the system is meant to be scalable under a single polling

master, a single operation with too many elements might take an unacceptably long time when multiplied by some number of clients. Optimizing the system to a reasonable response time is the single greatest challenge presented by Factorio's mechanics, and necessitates not only inventive wiring of algorithms but also finely trimming down the system protocol. Table 1 depicts synthesized logical elements and their minimum propagation time.

Logic Element	Components	Minimum Response Time
Binary Flag	2	2 Frames
Memory Cell	10	5 Frames
Linear Distance Algorithm	16	9 Frames

Table 1: Implementations of Logic Elements

A second consideration is the possibility of intermediate states appearing on a wire mid-calculation and causing unexpected reactions. While this can be prevented any number of ways, most require elements isolating the results from the main wire and therefore induce latency. In practice it is possible to precisely design around this within the clients, while assimilating the extra latency into the architecture of the server. The optimization necessary under these conditions provides fascinating insight into the design assumptions of realizable systems.

5. Bus Protocol

The scalability requirements of this project must be reiterated. A single server must support M stations and N trainyards, and K instances of the server must be overlaid to handle K types of resources. Supposing M, N, and K are kept within bounds of what could realistically be built within Factorio, this suggests less than a hundred of each client type per resource, and less than twenty types of resources. By laying a single global wire, we can thus allocate two wire variables to each resource, suggesting an additional separation of concerns. For each resource there can be two independent servers, one managing trains loaded with that resource and the other managing empty trains yet to be loaded with that resource. Since the servers only directly manage stations and trainyards, control of a given train will pass between these servers after each dispatch. Thus, K global wire variables must be allocated to the K resources, and an additional K global variables must be allocated to the absence of those same resources. Keeping K less than twenty, this can be achieved comfortably. Wire variable 0 is dedicated to "empty iron", wire variable 1 is dedicated to "full iron", and so on.

A single 32-bit wire variable must then transmit all server and client communications. In other words, we are essentially working with a 32-bit parallel bus. In order to reduce the number of circuit elements needed, values are kept between 0 and 2,147,483,647, and all messages must be issued in a single frame. To that end, the bus is multiplexed into four components: B, D, X, and Y, in the form B,DXX,XXY,YYY as follows:

B is a single bit, used as a binary command or interrupt.

D is a single digit, used to communicate the server's current mode.
X and Y are four-digit variables used to pass arguments on the wire.

Base-10 was chosen for readability and without loss of generality; X and Y should not need to exceed ten thousand in any realistic case. The various server modes indicated by D will trigger specific algorithms in receiving clients, and will be enumerated specifically in sections 8 and 10.

6. Client Internal Protocol

The clients, consisting of stations and trainyards, are reactive entities which push values to the global wire variable assigned to their resource in response to polling by the server. In order to minimize response time, responses to server queries are pre-computed and stored in local memory cells, and then pushed to the appropriate global wire variable immediately upon prompting. Internally clients are composed of several algorithm modules, several memory cells, and a wire spanning all modules that carries local variables.

Interfacing with the relevant global wire via a demultiplexer that broadcasts the global state to local variables, the client then consolidates a response variable that is pushed onto the global wire. The client is able to respond to server polling within only a few frames, so long as the output has been computed before polling. Ideally, this response time is only five frames, or one sixth of a second.

7. Server Internal Protocol

As a proactive entity that does not interact with instances of itself, the server performs administrative tasks that would incur too much delay to include in the clients. The most striking difference in how the server treats response time can be seen in a module that interfaces the server with the global wire. While the clients return simple responses in real-time, the server methodologically generates a command of set duration and saves it to a memory cell at the moment of broadcast. Server output, the message pushed to the relevant global wire variable, is then frozen for the duration of that message while the server goes to work constructing the next message. At the end of the current message duration, the interface automatically switches to broadcasting the newly constructed message and the process repeats.

The other major difference is that the server sets the current mode of operation and runs algorithms accordingly. Based on its current mode the server polls through the clients, executing other routines upon client responses. The various modes of operation, and the associated algorithms run by the server, are detailed in the next section.

8. System Modes: Routing

The value of the ninth digit D of each global wire variable indicates the current mode of operation, and therefore contextualizes the other variables B, X, and Y as follows:

0: Impossible state, implying that either some error has occurred or the client is not connected to a server. In either case, the client halts operation until it is configured into another server.

1: Polling of stations, the default state of the server algorithm. Y is the client name, B is an interrupt bit which the station client may set to request processing upon seeing its name appear in variable Y. This interrupt leads immediately to mode 2.

2: Pre-bid. X and Y are left blank by the server so that the interrupting station client can broadcast its coordinates. During this step, all trainyard clients compare the broadcast coordinates to their own and calculate the relative distance. After a brief interval, transitions to mode 3.

3: Bid polling. Y is the client name, X is the current lowest distance to the station coordinates. Upon seeing their name appear in Y, trainyard clients compare X to their calculated distance and, if appropriate, set B to interrupt. The trainyard client pushing a distance to X will acknowledge the interrupt by letting X=0, and upon seeing this acknowledgement the polled trainyard client will push its own distance to X. The process repeats as the poll iterates through all trainyards. On completion of the bid, transitions to mode 4.

4: Bid complete. The trainyard client with the current lowest bid, as well as the station client that initiated the poll, will release their train and enable their station respectively, thus dispatching the train to that station.

The remaining modes are used to assign unique names to clients, and will be listed in section 10.

9. Enumeration of Clients

There are two essential configurations a client needs to participate in the aforementioned polling. The first is coordinates, for the purposes of calculating relative distances. The second is a unique name, so that it can be polled by the server. If clients cannot be given unique names, there is no way to prevent bus collisions that garble messages. Therefore, upon connecting to the global wire a client must wait for the server to assign it a name.

An interesting consideration in this problem is the possibility of multiple clients connecting at the same time and then taking the first name offered by the server. To properly generalize the system, it must be possible to place identical clients simply by dropping instances of a single blueprint. These clients, however, will be fully identical because Factorio contains no method to automatically differentiate instances spawned from a blueprint, which can also be used if the clients are partitioned from the system and later reconnected. If any difference of state is established in the client's internal variables, this could be leveraged via a polling routine to give them different names.

These problems share a common solution. Factorio contains no inherent way for entities to know their coordinates, so a coordinate system must be laid manually beforehand. Since a spanning Green Wire and a rail network are also necessary, it makes sense to build the coordinate system into this infrastructure. Starting from a central point of [500,500], a grid of rails is established, with a tower carrying a Green and Red Wire along each rail. The Green

Wire is fully connected for real-time communication, while the red wire segmented with incrementing components along the axes and filters at intersections. Thus, the Red Wire at any given junction contains the coordinates of that junction, and connecting that to a client's local memory is a trivial matter.

In this way a consistent coordinate system is established, as well as a difference in internal state of otherwise identical clients. A name collision can thus be handled by the server via polling through XY coordinates, and the contested name can be assigned to the station with the lower coordinate value. The single caveat is that clients of the same type on the same server may not have the identical coordinates. A number of optimizations allow this polling to be done in a reasonable amount of time. With this requirement established, the remaining modes may now be listed.

10. System Modes: Enumeration

5: Name polling. Triggered infrequently, this mode iterates through the assigned names, which should be a sequence of consecutive numbers beginning at 1 due to the way names are assigned. Y is the currently polled name, X is a response variable to be incremented by any client currently using the name. If the server observes X=1, the polling continues because the name is assigned correctly. If the server observes X=0, the server sets the interrupt bit to invite unnamed clients to take this name and polls the same name a short time later. If X=0 is observed for a second time, no client has taken the name and the poll continues. If X>1 in any scenario, multiple clients are sharing a name and this collision must be resolved before regular operation can continue. In this case, clients with the polled name set a flag to participate in the next process and the server moves to mode 6. If, and only if, the poll of assigned names completes, the server returns to mode 1.

6: Name collision handling. XY coordinates are polled on X and Y, and the first client with the flag from mode 5 set to have matching coordinates sets the interrupt bit B to claim the name. Other clients with the same name will erase their names and wait for mode 5 to assign new names, thus resolving the collision. Various intermediate steps are used to complete the 9999x9999-step poll in a reasonable amount of time. Upon resolution of the collision, the server returns to the beginning of mode 5.

The full operation of a server is depicted below.

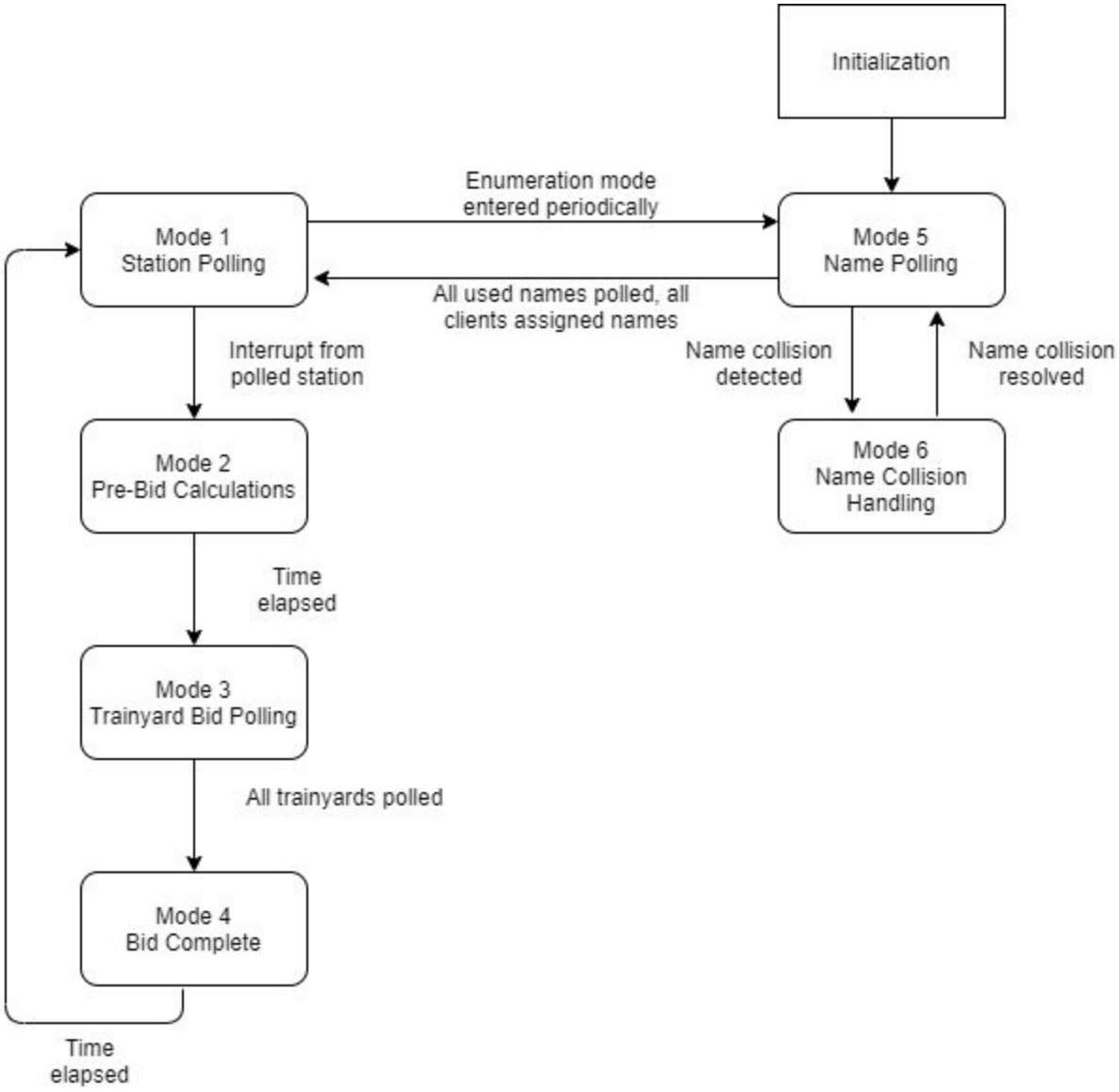

Figure 3: Server mode state-flow.

In order to reduce the duration of full polling cycles, there is incentive to split the names of station clients and trainyard clients into two namesets so that only the concerned client type needs to be polled in a given mode. To that end, modes 5 and 6 are specific to the enumeration of station clients, while modes 7 and 8 work identically but with regard to trainyard clients. Modes 5 and 7 are both triggered periodically by a timer in the server, with frequency determined by how much down-time is considered healthy for the system.

11. Implementation

To avoid grappling with the particulars of how Factorio handles circuits, substantive discussion of how particular algorithms were wired will be omitted from this paper. Additionally, the peculiar logic of Factorio circuits is not effectively conveyed either through diagrams or

screenshots. Given the problem definitions and solutions above, the short explanation is that the algorithms were implemented in circuitry, in architectures specifically designed to minimize the propagation delay and therefore total poll duration. System entities, compacted due to in-game space considerations, can be seen in the figures below.

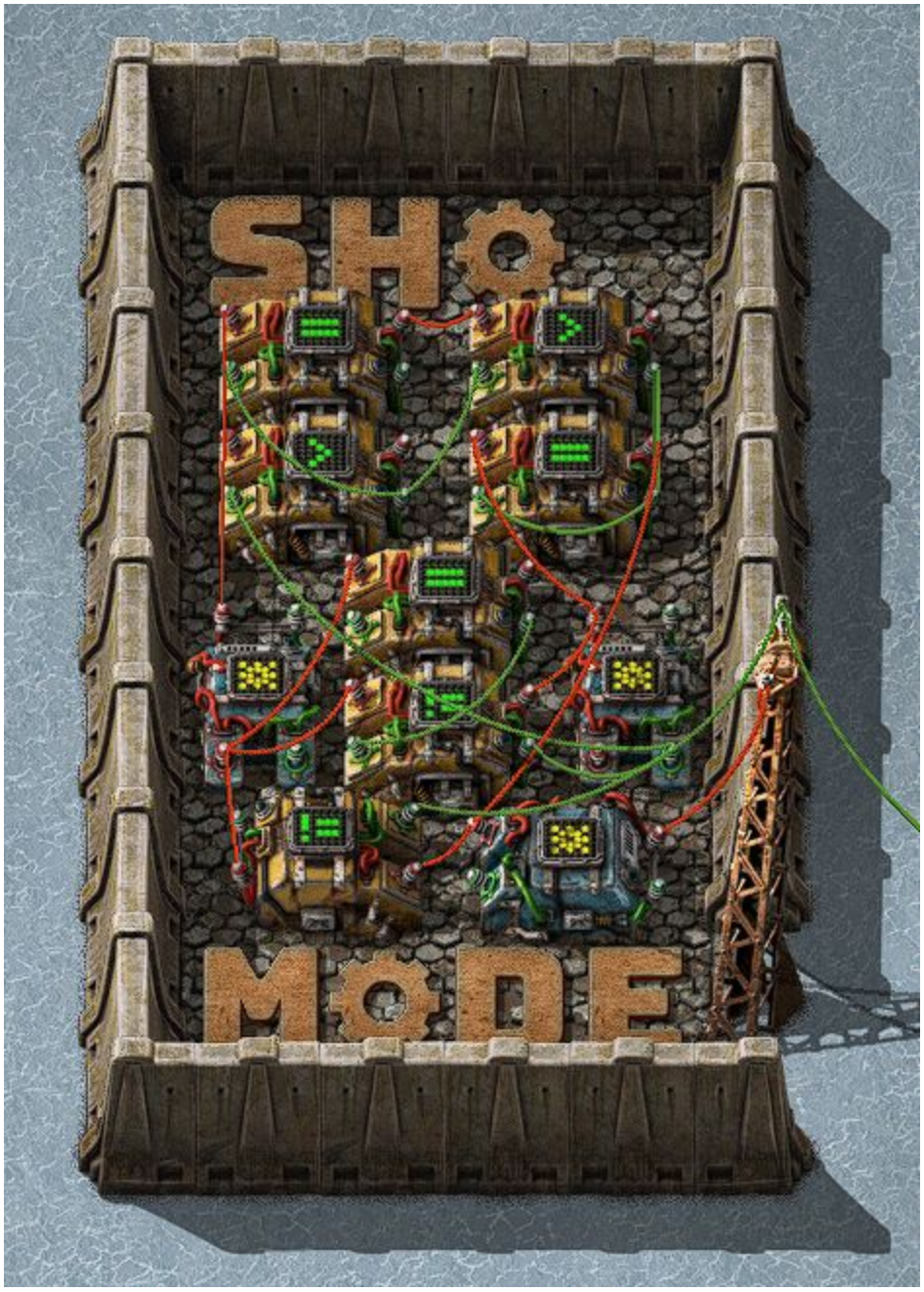

Figure 4: Memory cell implementation.

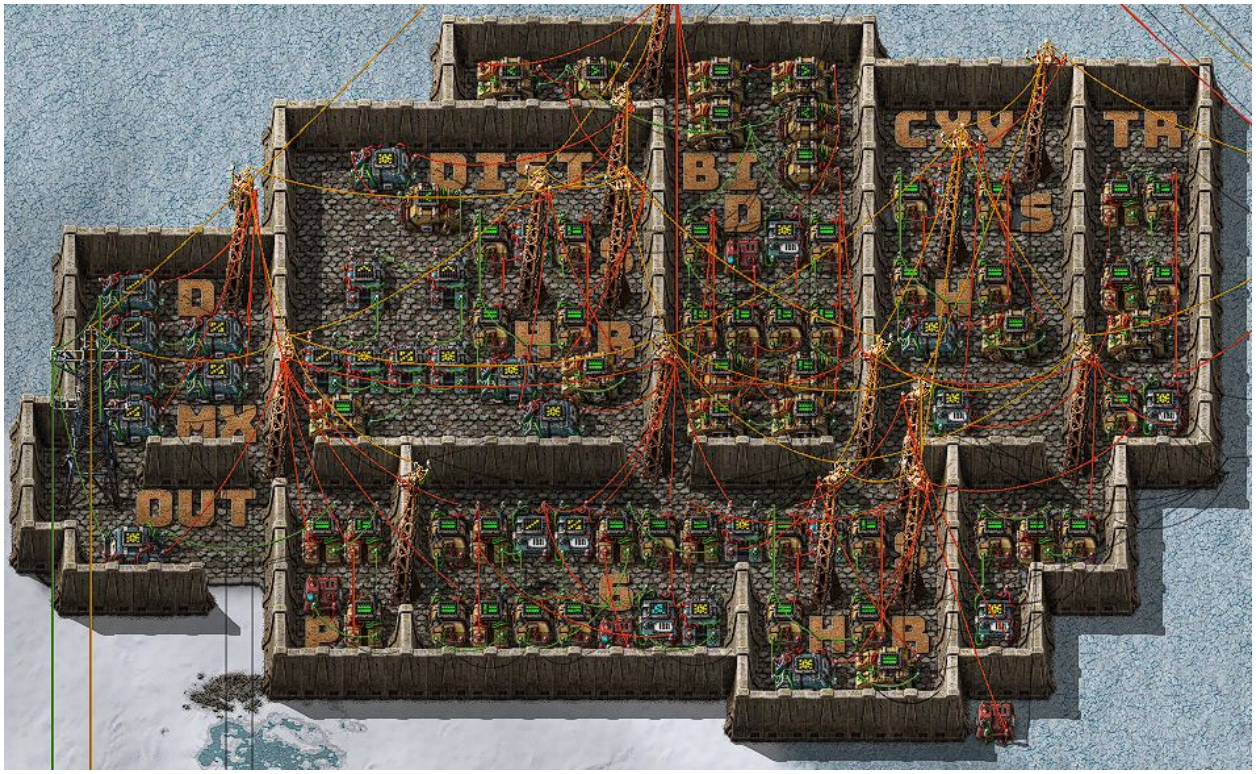

Figure 5: Trainyard client implementation.

Figure 6: Station client implementation.

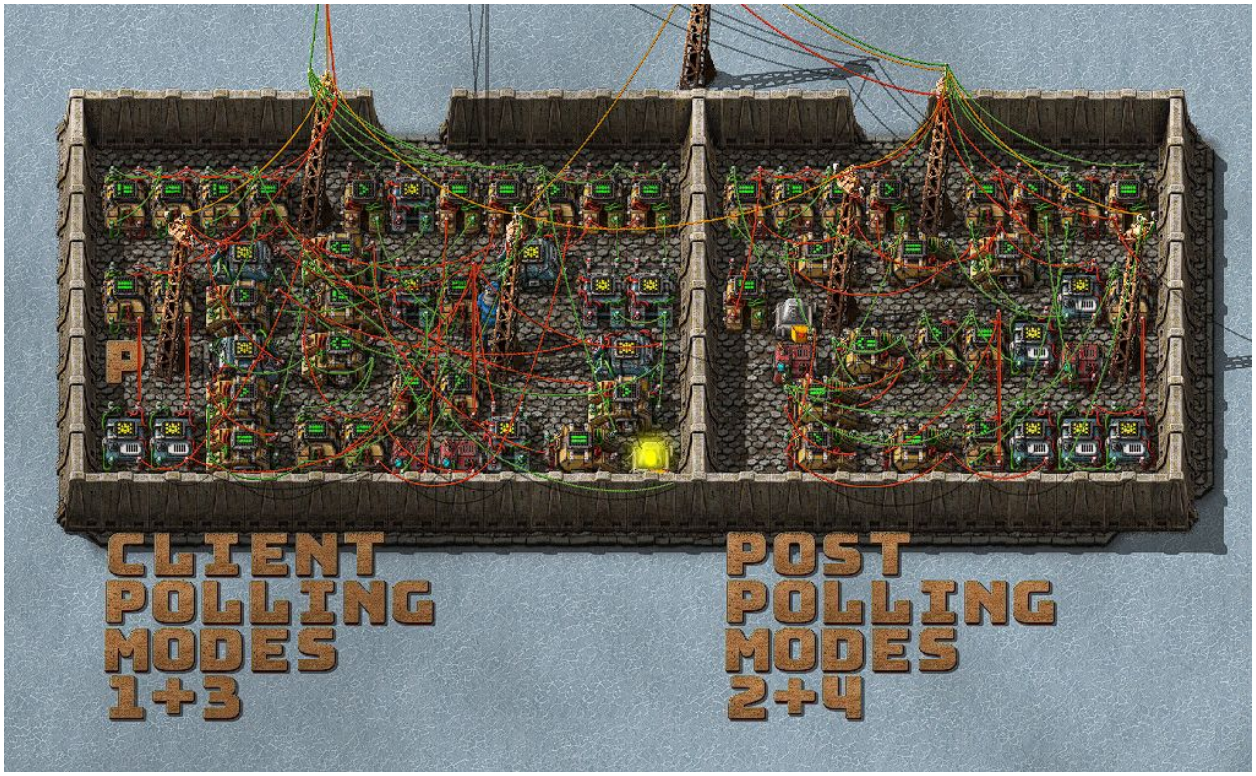

Figure 7: Partial server implementation.

Since modes 1 and 3 are similar polling algorithms that do not run at the same time, they share hardware to save space.

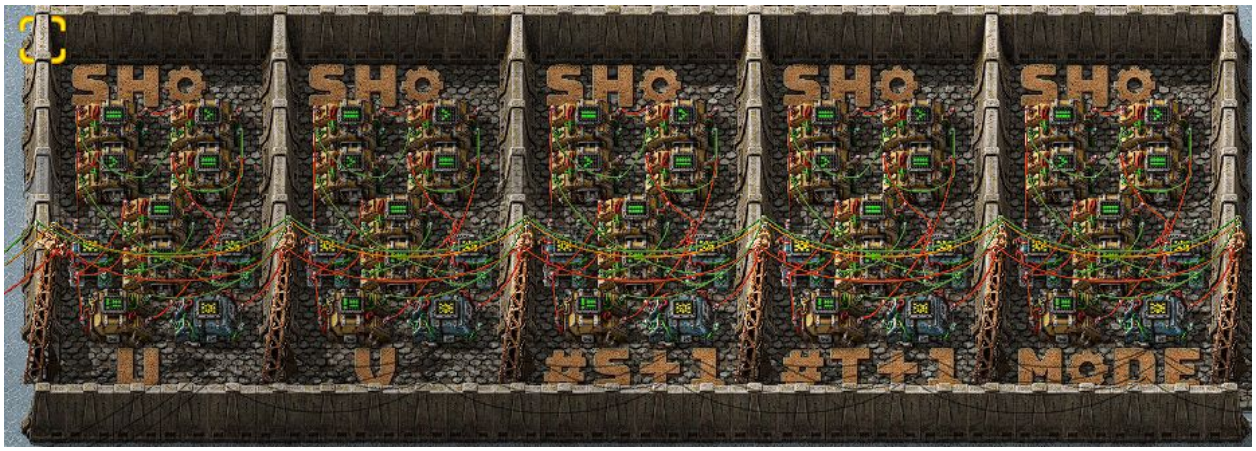

Figure 8: Server memory bank.

Certain modules have not been implemented. Since the protocol has been verified by applying mocked test cases to individual entities, full implementation and integration is not necessary to meet the analytical goals of the project. Though integration testing was successful when performed on the implemented modules, practical considerations have suspended any further implementation work. Physically wiring circuits in Factorio is a time-consuming process

that severely lacks the benefits of an IDE or debugger, and with the changes in version 1.1 there is very little benefit to fully realizing the system. The goal of this project, analysis of a familiar problem solved with alternate tools and constraints, can be performed solely with the verified protocol.

12. Comparison to Conventional Systems

When considering the concerns of a Factorio implementation versus a conventional distributed system, the advantages offered by an ideal communication channel cannot be overstated. Since communication in the chosen protocol is instantaneous and accurate over miles of wire, the true communication channel can be considered as the interfaces between the wire and system entities. This model of a communication channel, due to delays induced by the logical components, encounters some of the same issues as a realizable communication channel while the necessity of involving as few logical components as possible makes some conventional solutions unworkable.

In order to satisfy time constraints, entities must communicate in real-time without abstractions such as message packets. While the server commands and client responses are kept simple enough to be written directly to the wire, the fact that entities are expected to react immediately to signals on the wire means that verification and error prevention must be achieved through careful design and correctness proofs rather than physical countermeasures in the protocol. Despite the fact that these systems were implemented successfully, this suggests that the medium is unsuitable for serious projects because they would likely be difficult to maintain or expand long-term. Optimizing these circuits for speed and size is also mutually exclusive with architecture design best-practices, since separation of concerns is more difficult when modules are constantly acting on one another and even sharing hardware. A system less concerned with response time may be implemented in a more orderly fashion, but real-time considerations are an issue when components each require a full thirtieth of a second in propagation time.

In defense of these logical components, precise and error-free 32-bit digital communication is extremely useful. The message multiplexing protocol B,DXX,XXY,YYY allows for a straightforward communication scheme that is extremely convenient compared to the quantization or noise considerations of realizable bus wires. It is clear that this precision, along with the ideal communication channel provided by the wire, is the sole reason real-time distributed design is at all possible in the face of such intractable response-time concerns.

An exception to this model of interfaces-as-communication-channel is the consideration of a Byzantium situation. To lay out the parameters, suppose that at any point the global wire may be severed and the system may be segmented. Further, suppose that any client may be disconnected or destroyed at any moment, due to either player action or Factorio's game mechanics. Assume that the server is at the center of the coordinate grid and adequately defended. Since the server broadcasts some message at all times, clients can easily recognize such a partition and automatically disable themselves in a classic solution to the single-server Byzantium problem. If the partition is later mended, the disabled clients will not have names and will wait for the naming algorithm before resuming normal operation. This change in medium does not significantly effect how the Byzantium situation is handled. Alternate architectural

solutions to the Byzantium problem are available, often capitalizing on the fact that simulated hardware can be freely duplicated, but if we are willing to go that far it is simpler to just duplicate enough peripheral hardware to ensure segmentation faults never happen at all. For the sake of simplicity, handling the Byzantium problem with the same algorithm that provides names to new clients is a straightforward choice.

To summarize, building a real-time scalable distributed system in this medium provided options and challenges that drastically altered the details of the optimal protocol, but broader design patterns persisted across mediums. The available components ranged from ideal communications systems to terribly slow logical operators, resulting in a strange balance where the given problem was marginally solvable but more complex problems would be prohibitively difficult to implement and maintain.

13. Physical Realizability

Having developed this particular solution to a real-time control problem, this section will consider the feasibility of that algorithm in physical media. To match the problem background, the physical system under consideration will be a controller for a model railroad. This system will try to match the response-time parameters of the Factorio system using a similar protocol and minimal hardware. It should be noted that while the Factorio system is built atop Factorio's train routing algorithm, which relies on instantaneous communication between trains and distant stations, the Factorio system controls its clients so precisely that this capability is not crucial. Supposing that the realizable system handles routing in a similar way, that is trains need only be dispatched to a given station and will adaptively determine a route themselves, a protocol similar to the Factorio system can be implemented.

Using a 32-bit parallel bus coupled with an appropriate error-checking method, messages can be shared across the realizable system in the same way as the Factorio protocol. This communication will not be instantaneous, but with client and server algorithms executing on physical circuits rather than artificial components with collectively high propagation time, the realizable system should still be faster. Any optimized digital system capable of executing the algorithms described should be able to surpass the arbitrarily high response times of the Factorio implementation, especially sequences of simple operations that would take a full thirtieth of a second in Factorio. Thus, the timing considerations of the realizable system are determined by both the response times of entity modules and the time needed to communicate across a realizable channel. Both can clearly be made much smaller than the response time of the Factorio protocol, suggesting that such a protocol can be implemented with realizable hardware along with verification hardware that was not needed or practical in the Factorio protocol.

The question, then, is how optimal the realizable system is. Because the wire mechanics in Factorio suggested the specific 32-bit parallel bus format, this architecture was optimal for Factorio. Without that restriction on the realizable system, there are far more hardware options available, and far more viable architectures other than this customized single-master CANbus system. It would be an enormous coincidence if the particular solution for the Factorio system turned out to be optimal for the model railroad system. Indeed, protocols that do not rely on a single master would scale more readily and allow more elegant solutions to Byzantine problems.

Architectures other than a 32-bit bus could achieve the same results with lower hardware cost. Indeed, the Factorio architecture was a specific solution to unusual parameters and this is reflected in its suitability under other parameters.

Conclusion

The purpose of this project was to examine how an alternate medium would influence the design process of a given class of system. As expected, undertaking a familiar design problem with unconventional tools in an unconventional medium resulted in unusual considerations that produce a solution specific to those considerations. Despite the presence of an ideal digital communication channel, the broad concepts of the discovered solution mirrored those of realizable options with certain low-level components changing roles. This paradigm may be specific to the medium chosen; a more permissive medium might allow architectures that bypass core concerns of distributed system design, while a more restrictive medium might result in a very different protocol or even no solution at all. The tools offered within Factorio straddle a strange balance between these extremes, offering an ideal communication channel but also making systems with low response-time extremely difficult to maintain or expand. This is because keeping response-time down means solving many practical challenges the hard way instead of through abstraction. The system optimized for Factorio can be physically realized with minor changes, but is in no way optimal outside of that specific medium.

This project illustrates the enormous medium-dependence of system implementations. Had a conventional distributed system protocol been wired in Factorio verbatim it would have been prohibitively slow. Constructing a custom solution from the axioms of the medium while importing only high-level design concepts results in a far more optimized system.

Appendix 1: Train Behaviour

1) Each individual train can be configured with a sequence of stations to visit, and conditions (handled by the system) which must be met in order to leave that station.

i) Once a train meets the conditions to visit a station, the actual routing will be handled by the game engine, so the system is only concerned with the selection of stations.

ii) If multiple stations of the same name exist in the world, the train will visit the nearest and then continue to execute the schedule. Thus there is no change in behaviour caused by distant stations with the same name.

2) Stations can be enabled and disabled by the system.

i) A train will not visit a disabled station for any reason.

ii) If the next station in the schedule is disabled, the train will ignore that step and continue to execute the schedule.

iii) If there are multiple stations of the same name, the train will route to the nearest station that is enabled.

iv) If all stations in the schedule are disabled, the train will idle.

v) If a train is en route to a station that is suddenly disabled, the train will automatically reroute.

vi) If a train is en route to a station but a closer station with the same name is suddenly enabled, the train will automatically reroute.

vii) A train already idling at a station (either due to not meeting the system's condition for leaving, or not having any enabled destinations) does not care whether the current station is enabled or disabled.

3) Given that the rail is laid correctly, trains will consider one another while pathing so as to avoid collision. However, there is no limit to the number of trains that may path to one station.

i) Queues at a station cannot resolve until each train has accessed the station, which potentially leads to deadlocks.

ii) In an architecture where many trains are idling and waiting for a target station to be enabled, all such trains will immediately attempt to visit a station that suddenly enables, causing a queue and most likely a deadlock.

iii) Version 1.1 of Factorio implemented a tool to limit the number of trains en route to a given station. Since this paper is concerned with the version 0.17 branch, this feature will not be used. The extent to which this feature would simplify the problem is left as an exercise to the reader.

This is the context in which our control system must be built. 1 and 2 show that the system is only able to interact with the train network in two ways: enabling and disabling stations, and giving trains permission to leave their current station. 3i and 3ii illustrate the necessity of a precise control system if the train network is to scale and run without user intervention. Since it has not been explicitly stated yet, I will add that the end goal of this system is to run the factory autonomously without any sort of player involvement.

As an aside, this discussion assumes stations are designed for one and only one train. While this is unrealistic in practice, such a model can easily be generalized if larger stations are considered as multiple instances of a single station. More refined methods of handling large stations have been implemented in the project, and are necessary due to the coordinate-sharing restriction in section 9, but are beyond the scope of this discussion.