The MvK client-server architecture

Yentl Van Tendeloo

December 16, 2014

1 Introduction

This report introduces the client-server architecture that is used in the new version of the Modelverse Kernel (MvK) [7]. Whereas this currently has an explicitly modelled front-end, the HTTP communication between this front-end and the Modelverse server was not yet explicitly modelled. In this work, we explicitly modelled the network communication: the behaviour is modelled using Statecharts, and a performance analysis is performed using DEVS. While this explicit modelling offers us the advantages associated with modelling and simulation, we are able to circumvent problems that would arise when using out-of-the-box network libraries.

1.1 Motivation

Up to now, the Modelverse server and (explicitly modelled) front-end relied on the HTTP libraries included by Python. These HTTP libraries however, are not a perfect match with an explicitly modelled front-end. Some of the disadvantages include:

1. *Blocking*. Frequently, default HTTP libraries require blocking communication. This makes it a bad match with statecharts, as this would block the statechart runtime from correctly processing timeouts and interrupts. Several non-blocking operations are supported too, though these are not a perfect match with statecharts either, as these should raise statechart events.
2. *Non-portable*. One of the advantages of statecharts, is that a compiler can be used to generate code for different platforms. By using language-dependent libraries, this advantage is mitigated, as the compiler would require knowledge about these libraries in all kinds of languages that it supports. Additionally, each library has its own interface, with its own semantics for every operation. While this is managable for a single language, this puts a huge burden on either the compiler writer, who has to take into account all different libraries and provide a uniform API, or on the modeller, who has to write a different model for each and every language that will be used.
3. *Single user*. Most of the default libraries that are included by default, only allow for a single user to connect. Though frequently there is also a multi-user version available, it is often unknown to the user how this is implemented internally. Whereas the threading approach is often sufficient, some platforms do not support threading in the way that it is implemented by the library. For example, in a TkInter based application, multi-threading is not supported. All threading has to be emulated using the TkInter event loop. Clearly, matching the default multi-user HTTP server to this approach is likely to become a problem.
4. *High abstraction*. The provided HTTP libraries, both for the client and server component, are frequently at a high level of abstraction. Sometimes, more information is known about the communication that will be going on, which allows us to optimize some parts of the code. For example when transferring huge amounts of data, it is possible to increase performance by increasing the receive buffer of the socket. Certainly if intensive processing is required for every receive, this is can speed up processing with an order of magnitudes. Another example is the use of socket reuse for HTTP connections. HTTP/1.1 supports the (optional) reuse of HTTP connections[4], though whether or not this is used is dependend on the library in use. Even though it is often possible to configure, or manually tweak, these libraries to include these specific optimizations, this is library dependend and again not easily portable.

5. *Black box*. Though HTTP libraries are easy to use and often quite efficient, their correctness is mostly only determined through the use of testing. Explicit modelling of the front-end offers advanced methods of model analysis, but this analysis would then stops at the level of the HTTP client (or server).

While external libraries exist that try to solve these problems, they are limited to a specific set of languages. Additionally, these libraries introduce dependencies to our applications, that are out of our control.

Concerning the performance analysis, we gain lots of advantages, associatied with modelling and simulation. In this work, we mainly focus on the possibility to evaluate the performance of a variety of situations. This offers us the following advantages:

1. *Cost*. Without an explicit performance model, the only possibility to measure the performance under a certain situation is to replicate that simulation in real life. Some of these situations are quite costly to reproduce, such as the performance that could be expected from the acquisition of a new (costly) computer. If a variety of computers is to be simulated, it is required to acquire these different types and run the associated operations on them. With an explicit performance model, only a performance model of that specific machine is required to allow for this analysis.

2. *Time*. Depending on the set of operations that is being studied, simulation is more efficient as it is not limited to the wall clock time. When simulating the behaviour of a single client, most of the time will be spent waiting for another request to come in. If this client is configured to only make a request every few seconds, the experiment will take a long time, mostly spent idling. On the contrary, simulation is able to progress through time at the maximum pace that the machine supports. Additionally, computationally intensive operations can be replaced by only a time increment, without wasting precious computational resources on the actual operation.

3. *Hypothetical situations*. While it is still possible to acquire the system under study if it is currently available (and affordable), this is not the case for hypothetical systems. Simulation makes it possible to analyze the performance of systems that still have to be built. This does not only apply to computational resources, but also to the network. Like what would be the performance of the system if network latency were to be reduced to exactly $0ms$, or what if it were extremely high? These situations are very difficult or impossible to replicate in real life, but offer valuable insights in the performance of the system in a variety of circumstances.

In [6], the performance of a distributed DEVS simulator was analyzed in a similar way, offering insights on the performance in (difficult to replicate) situations.

1.2 Goal

Our client-server architecture has several goals. First, there are requirements as to what operations should be supported. Second, our approach tries to deal with some of the disadvantages mentioned in the previous section.

Our requirements consist of:

- Clients should be able to issue HTTP POST requests, for which the MvK server sends an HTTP reply back.
- Server-initiated communication should be supported as a basic operation.
- Multiple users, with possibly interleaving operations on the MvK, should be supported. As the MvK currently does not yet support parallel operations, operations are sequentialized.
- Every connected client should have a server-side statechart, indicating its current modal state with the MvK. This statechart is able to receive a single, composite request from the client, and can split it up in multiple smaller CRUD operations on the MvK.

Furthermore, the following disadvantages that were previously identified will be circumvented:

1. *Blocking*. The new architecture is purely written in statecharts (with a small, necessary wrapper). Therefore, all operations are perfect for interleaving with other statechart components.

2. *Non-portable*. As all behaviour is modelled using statecharts, code can automatically be generated for a variety of platforms. Sockets are a common basis that is available in most languages, all of them offering (nearly) the same interface. Socket communication will still need a small wrapper that wraps the sockets to allow for statechart events to be processed. This wrapper is minimal though, as the supported events provide a perfect mapping with the available socket operations. Note that currently, there is no complete support for language-independent action code, so there will still be some manual changes that need to be done.

3. *Single user*. Due to the automatic interleaving offered by statecharts, and the extensions from SCCD which offer the dynamic structure of the statechart, multiple users are natural to support. Because every user gets its own statechart objects, they are automatically interleaved by the statechart compiler.

4. *High abstraction*. Our approach works on the lowest level offered by the language, being sockets. These sockets can be configured in the statechart. As an added extra, the socket communication is done in statecharts, making it easy to add some specific information into the communication.

5. *Black box*. We will offer up efficiency for the possibility to analyse the system. Analysis of the front-end no longer stops at the network libraries, but goes down to the low-level sockets. Even though we state that we lose some efficiency, the lower abstraction can more than make up this loss.

Additionally, code should be automatically generated from the behavioural model. Using a performance model of the architecture, performance analysis can be executed on the architecture.

1.3 Used formalisms

This section introduces the different formalisms used in the remainder of this report.

1.3.1 Statecharts and Class Diagrams

The Statecharts formalism was introduced by David Harel [5]. Statecharts is an extension of state machines and state diagrams with hierarchy, orthogonality, and broadcast communication. It is used for the specification and design of complex discrete-event systems, and is popular for the modelling of reactive systems, such as graphical user interfaces. A Statechart generally consists of the following elements:

- states, either basic, orthogonal, or hierarchical;
- transitions between states, either event-based or time-based;
- actions, executed when a state is entered and/or exited;
- guards on transitions, modelling conditions that need to be satisfied in order for the transition to "fire";
- history states, a memory element that allows the state of the Statechart to be restored.

In the remainder of this report, we use the SCCD formalism [3], which is a combination of Class Diagrams and statecharts. Every class diagram has an associated statechart which defines its behaviour. Just like new objects can be instantiated, new statecharts can be instantiated too. Furthermore, all of these statecharts are able to communicate with each other by raising events.

1.3.2 Parallel DEVS

DEVS [9], and in particular Parallel DEVS [2], is used to model the behaviour of discrete event systems. Its basic building blocks are *atomic DEVS* models, which are structures

$$M = < X, Y, S, \delta_{int}, \delta_{ext}, \delta_{conf}, \lambda, ta >$$

where

- The *input set* X denotes the set of admissible inputs of the model. X is a structured set $X = \times_{i=1}^{m} X_i$ where X_i denotes the admissible inputs on port i.
- The *output set* Y denotes the set of admissible outputs of the model. Y is a structured set $Y = \times_{i=1}^{l} Y_i$ where Y_i denotes the admissible outputs on port i.
- The *state set* S is the set of sequential states.
- The *internal transition function* $\delta_{int} : S \rightarrow S$ defines the next sequential state, depending on the current state.
- The *output function* $\lambda : S \rightarrow Y^b$ maps the sequential state set onto an output bag.
- The *external transition function* $\delta_{ext} : Q \times X^b \rightarrow S$ with $Q = \{(s,e) | s \in S, 0 \leq e \leq ta(s)\}$ gets called whenever an *external input* ($\in X$) is received.
- The *time advance function ta* $ta : S \rightarrow \mathbb{R}^{+}_{0,+\infty}$ defines the simulation time the system remains in the current state before triggering its *internal transition function*.
- The *confluent transition function* $\delta_{conf} : S \times X^b \rightarrow S$ is called if both an internal and external transition collide at the same simulation time, replacing both functions.

A network of atomic DEVS models is called a *coupled DEVS* model. Output ports of one atomic DEVS model can be connected to one or more input ports of other atomic DEVS models using "channels", defining a *transfer function* to translate output to input messages. Parallel DEVS is closed under coupling, which means that coupled models can be nested to arbitrary depth.

2
Behavioural model

As a first step to the modelling of the network, the behaviour is explicitly modelled using SCCD (Statecharts + Class Diagrams). This step comes before the performance optimization, as we will be using this model to calibrate the performance model, and to validate it afterwards.

2.1 Requirements

The communication protocol should provide support for several operations, which should be encoded in the form of an SCCD model. All communication should happen through a standardized protocol, in our case in the form of XMLHTTP requests. The basic use case is a single client request, which gets answered by the server with a single reply. To support all our operations, there should additionally be support for:

- *Server initiated communication*: Some operations on the MvK will require the MvK to send requests to the user. As this is not natively supported by XMLHTTP requests, support has to be added to emulate this kind of behaviour. The user of this network client/server should not be concerned with the underlying architecture (being XMLHTTP requests), so there should be support for both directions of communication that is transparant to the user.
- *Multiple users*: Due to the requirements for the MvK itself, multiple users should be able to simultaneously connect to the same server. All complexity concerning multiple users is handled by this network component, which will map all this complexity to events being generated.
- *Client statechart*: In the normal request/reply architecture, the server is stateless. The MvK supports composite operations, such as model transformations, which are composed of multiple smaller steps. In this context, the server should thus have some state. It is again most logical to implement this in statecharts. Therefore, there should be a statechart for every client, which receives only messages from its own client. That same statechart should interract with the MvK, providing a small wrapper for the client.
- *Management statechart*: In multi-user scenario's, there should be some kind of management concerning who gets some processing time. This should again be implemented in a statechart, which receives the requests and replies from all clients and orders them in the order it sees fit. Some of the possibilities of this approach include queueing, transaction management, and priority.

2.2 Assumptions

Even though we want every piece of the system to be modelled in SCCD, there are some limitations to this approach. The most important limitation in our context is the lowest level, where there needs to be a mapping to the low-level sockets. As these operations do not work together perfectly with statecharts, all socket operations were extracted to a 'socket library'. This socket library interacts through the use of statechart events. For example, sending a message on a socket gets mapped to the sending of an event, containing the socket and data to send. This event is catched by the socket library, which will (on a seperate thread) invoke the necessary commands on the socket. Sending received data from the socket is also done using events, where the socket library injects statechart events into the running statechart. This approach allows us to use the full power of sockets in combination with the use of statecharts.

Because the MvK itself does not have a model part, it is simply put into the statechart as a 'black box'. In the sense that it processes a single event, and raises the request immediately after.

2.3 Model

An overview of the complete model is shown in Figure 2.1.

It consists of the following classes (with associated behaviour):

- *Socket* consists of 4 orthogonal states:
  1. Either the socket is still actively listening for input, or it is closed. Note that this closed state is different from the final state, as a socket can be closed in one direction. Even though the socket containing the request is closed, this does not mean that a reply is not required.
  2. The socket waits for data to arrive in its queue, and sends the data as soon as data arrives. This sending is done in a different state, as it is possible for the send to fail after a certain number of bytes is put on the socket. In that case, the send is retried, but only of the next bytes. As soon as all data is sent, the queue is again checked for new messages.
  3. Incoming events that should be sent on the socket are queued by the socket, as it is possible for them to arrive at a higher rate than the socket can process them. Furthermore, we don't want messages to interleave each other (e.g. due to partial sending).
  4. On the receiving side, the data that is received from the socket is parsed until a complete header is found. If the header is complete, the content length property is extracted and used to determine the length of the payload. Waiting for the payload then consists of accumulating incoming data until the length exceeds or equals the content length identified in the header.
- *Server* consists of 6 orthogonal states:
  1. The queue will process messages coming from the different sockets. They are queued, as multiple sockets can send data in parallel.
  2. This orthogonal state waits until some data has entered the queue. As soon as this happens, the data is processed by deserializing its contents. Should deserialization fail, an error is send back immediately. Otherwise, the deserialized data is forwarded to *Statecharts* and the operation blocks for a reply. This blocking until a reply is received is currently still necessary, as the MvK itself does not support multiple users at the same time. Therefore, every request implicitly starts a critical section, which is handled as an atomic operation. Consequently, no queue is necessary for the replies.
  3. A queue is necessary for server-initiated communication, as these are send at the discretion of the *Statechart*.
  4. Another orthogonal component checks for data in this server-initiated communication queue, and waits until a long polling request is made for that specific client. Should such a request already be done (as it should be in most cases), the request is send immediately. Otherwise, the operation blocks until the client has made such a request. Due to the design of HTTP, which doesn't allow for native server-initiated communication, we can only wait until the client decides to issue such a request.
  5. Yet another orthogonal component corresponds to the logic required to start up a server socket. This orthogonal component will bind the socket to the desired port and start listening on it. Connections are accepted when they are issued, which causes a new *Socket* to be created.
  6. Finally, one orthogonal component is responsible to catch close events sent by the different *Socket*s. Upon receiving a close event, the corresponding *Socket* is deleted.
- *Controller* is a fairly simple statechart at the moment, as it does not yet include any kind of management logic. All it does is set up the different objects that are required for communication. Internally, it is used to route messages to the differrent objects.
- *MvK* is the statechart containing the actual MvK. As the MvK itself is not modal, this is simply a wrapper around its interface, translating all events into operations. The reply returned by the MvK is raised as an event too.
- *Statecharts* is again a simple statechart, which waits for a request and forwards it to the respective *Statechart*. If the *Statechart* corresponding to the statechart does not yet exist, it is first instantiated.
- *Statechart* is the client-specific statechart. This part should contain most modal parts about the MvK, as this has to be handled on a client-by-client basis. The basic statechart simply waits for a CRUD request and forwards it to the MvK. An orthogonal component then waits for a reply from the MvK and wraps it into the required form for serialization. Future versions of this statechart can make use of the server-initiated communication by sending out the necessary events.

A client-initiated request is processed in the following order:

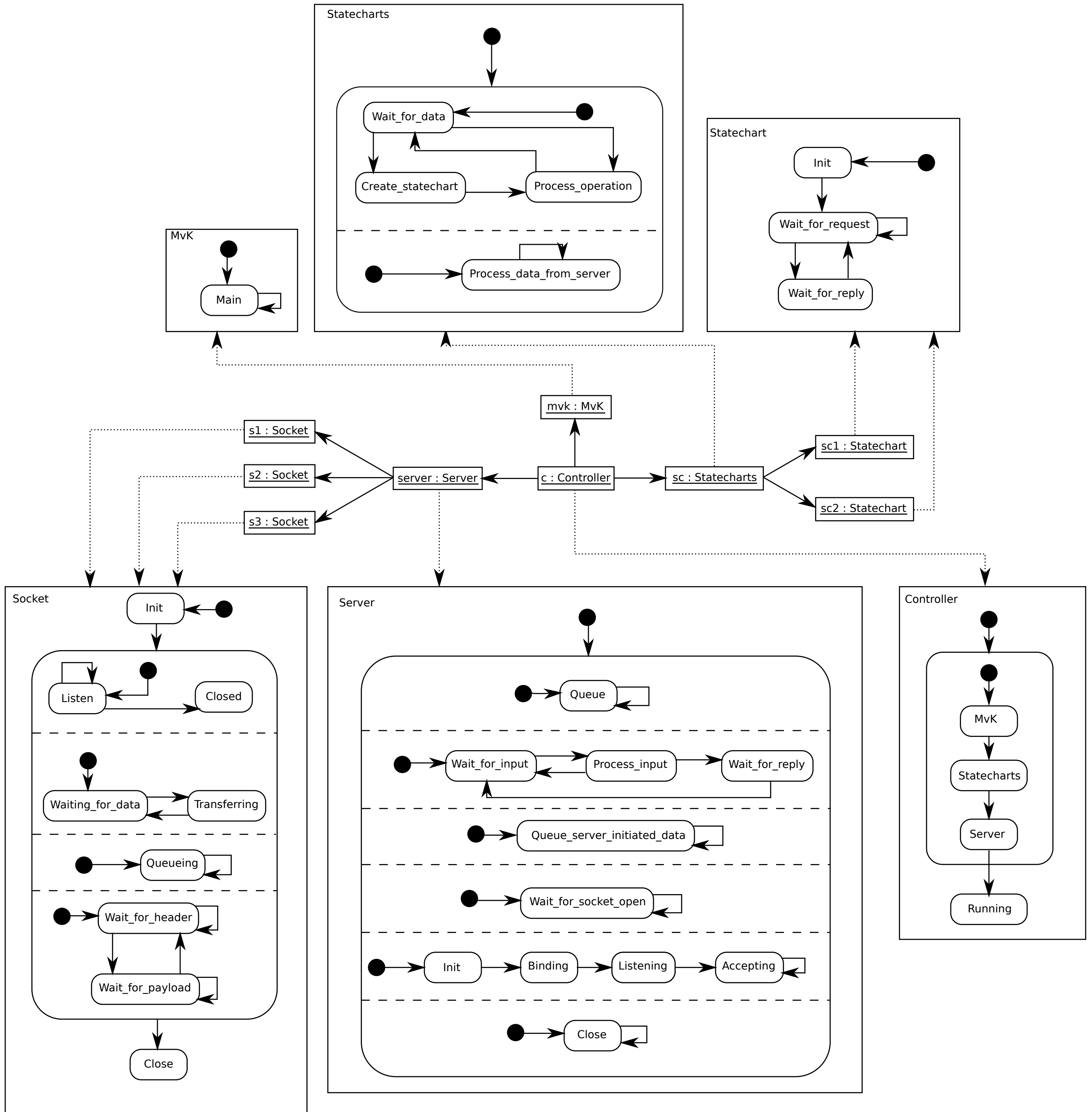

Figure 2.1: Overview of the architecture in SCCD

1. The *Socket* statechart receives an event from the 'socket library'. It unwraps it from the POST data, decodes the enclosed data, and determines how it should be forwarded to the next statecharts (i.e. is it a request or a reply).
2. The *Server* receives the event from the Socket and deserializes the JSON content that is enclosed. Should a deserialization error occur, this message is marked as invalid. After deserialization, the message is forwarded to the next statechart.
3. The *Controller* receives the event and processes it according to its managing rules. Depending on e.g. a running transaction or the current system load, it is decided when the event should be propagated. As soon as the event is allowed to be processed, the event is forwarded.
4. The *Statecharts* receives this event and decides to which statechart this should be processed. Because there is no 1-to-1 mapping between sockets and statecharts, all events contain a unique client identifier. If this is the first time that this specific identifier is seen, a new *Statechart* is started. The event is now forwarded to the specific *Statechart* that is responsible for the enclosed client identifier.
5. The *Statechart* receives the event and processes it according to its current state. For CRUD requests, these events will simply be forwarded to the MvK directly, but composite operations might be expanded. It is also possible that the received data was a reply to a server-initiated request, in which case the input is processed directly.
6. The *MvK* receives requests from all running statecharts, and will answer them in FIFO order. After an event is processed, it raises a new event containing the response.
7. The *Statechart* catches the event with the reply, and depending on its state, it forwards the reply to the client, or it processes this reply itself. In case the reply is forwarded, it is directly forwarded to the server.
8. The *Server* serializes the content in the reply (or server-initiated message) and decides on the appropriate socket to send this data to. For normal replies, the message will be put on the socket that caused the original request. For server-initiated communication, the message is put on the socket that is still waiting for a reply (long polling).
9. The *Socket* finally receives the data and turns it into a form that is appropriate for a reply to the running POST request. The final result is directly sent on the socket, by outputting an event that is understood by the socket library.

For a server-initiated request, the following steps are done:

1. The *Statechart* issues a server-initiated request by sending the corresponding event.
2. The *Server* catches this event and places it into a queue. An orthogonal component waits until a long polling request is done by the client. The event is further processed as soon as a socket is open that was used for long polling.
3. The *Socket* transmits the data like usual, putting the POST request data in the data field of the HTTP reply. This is then processed by the socket library like a normal reply.
4. Finally, the client has to redo the long polling request, to make sure that the server can again initiate communication.

Due to the choice for long polling, server-initiated communication has a relatively large overhead, as the client has to initiate a long polling request before the data can be sent.

3 Performance model

As a second step, the performance model was created based on the architecture that we created in the modelling of the behaviour. This phase consists of several steps: modelling, parameter calibration, validation, and finally analysis. All these steps are discussed in the remainder of this chapter.

3.1 Requirements

It is always required to first identify the requirements for the model. This way, it is known which abstractions can be made in the model, without invalidating the results.

For our model, these requirements are in the form of the parameters that we want to tune, and the results that we want to get out. Our tunable parameters are:

1. *Server processing speed*. The MvK server will most likely be the performance bottleneck. Therefore, the time it takes for the server to process a single request is a mandatory parameter. Some more detailed parameters are contained in this single parameter, such as the time required in each layer defined in the previuos chapter. As these parameters do not have much individual influence, they are accumulated. Additionally, measuring these parameters at such a fine-grained scale becomes much more difficult.
2. *Client behaviour*. Different types of clients cause different loads on the MvK. Some of the supported client types for the MvK are fully automated clients (possibly issueing multiple asynchronous requests), manual clients (issueing a single synchronous request, with a big delay between requests), or a user-guided automated client (sending a batch of requests automatically, followed by some delay for the user to choose the next request).
3. *Network latency*. Network latency is a defining factor for the client behaviour of non-automated clients. For non-automated clients, the network latency will be added to the total duration of their request time, but also to the absolute time of their next request. On the other hand, automated clients issue asynchronous requests and therefore their behaviour is unrelated to the network latency. For example, a non-automated client will not be able to make more than one request
4. *Number of clients*. As the final goal of the MvK is to serve a multitude of users, the number of clients is a vital configuration parameter. Intuitively, increasing the number of clients will have a serious effect on the time requests take, so this parameter needs to be present.

For each of these cases, we are only interested in how long it takes for the client to receive a reply to its request.

3.2 Assumptions

Related to our requirements, several assumptions can be made during the abstraction and simplification of the model. The most important assumptions that are made, are:

- *Server-initiated communication* has no influence on the performance of simple requests.

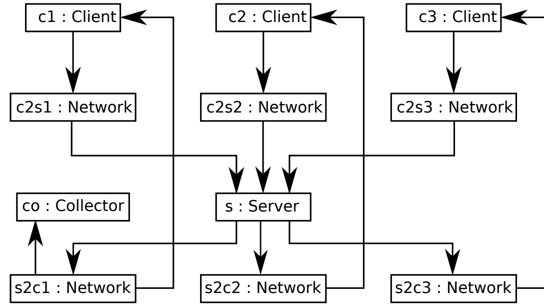

Figure 3.1: Structure of the DEVS model for 3 clients

- *Queueing* is only done when the MvK is still processing a previous request. Related to this, no client has priority over the other, and all queues are FIFO.
- *Non-preemptibility* of the processing of a request is assumed. This is certainly not the case in real situations, as every step in the statechart allows for preemption. However, this is assumed not to have a significant impact on performance.
- *Network bandwidth* is assumed to be infinite. For the MvK, we know that requests and replies are simple JSON-encoded data, which are relatively small compared to currently available bandwidths.
- *Packet loss* is assumed not to be a problem, as TCP/IP is used. While this combats packet loss, packet loss still introduces higher latency due to timeouts and resends. All of this will be captured, as network latency is not assumed to be constant.

3.3 Model

A graphical overview of the structure of our model is shown in Figure 3.1. A detailed description of each atomic DEVS model follows:

- The *Client* atomic DEVS model is responsible for the sending of requests. Its implementation defines the client access behaviour mentioned previously.
- The *Network* atomic DEVS model causes the introduction of latency into the simulation. Before the client's request is received by the server, it has to pass through the network, which inevitably adds a certain amount of latency. As we have chosen to ignore the bandwidth limitations, and consequently queueing, this component only adds a small delay that is independent of the message being passed. Packet loss and out-of-order delivery is also ignored, making this a very simple model.
- The *Server* atomic DEVS model implements a basic FIFO queue, which will queue all incoming requests until they can be processed. Because the actual results of the request are not important in the simulation, the request will not be evaluated (thus saving time), but processing time is introduced.
- Finally, *Collector* simply receives all messages that are sent back to the client and determines the time it took for a response. These values are saved into a list, which is later used for for the computation of statistics.

Note that there is only a single shared resource: the *Server*.

Our model takes three different configuration parameters:

1. *Processing time distribution*, which determines the distribution that should be used by the *Server* to compute the *timeAdvance*.
2. *Network latency distribution*, which determines the distribution that should be used by the *Network* to compute the *timeAdvance*.
3. *Number of clients*, which determines how many clients are connected at the same time.

The client behaviour is also a configurable parameter, but to support complex behaviour patterns, the client behaviour is just represented as a complete atomic DEVS model.

3.3.1 Calibration

After modelling the system in Parallel DEVS, calibration is required for subsequent validation. When calibration is finished, the system for which the model is calibrated can be simulated in a variety of situations. In our case, we first start by calibrating the model to our own machine. This same calibration can be done on any machine that needs to be simulated. In case a hypothetical

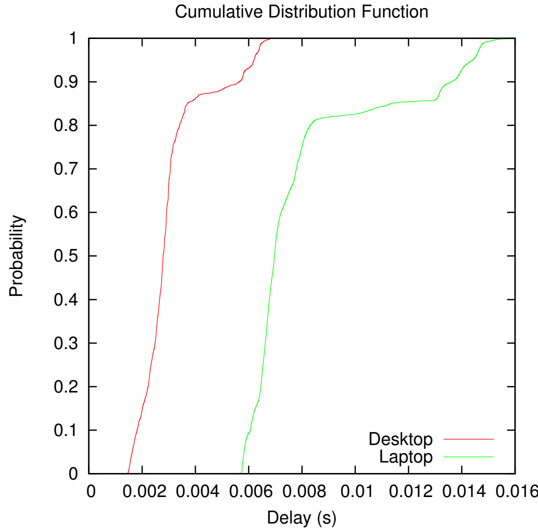

Figure 3.2: Measured server processing time

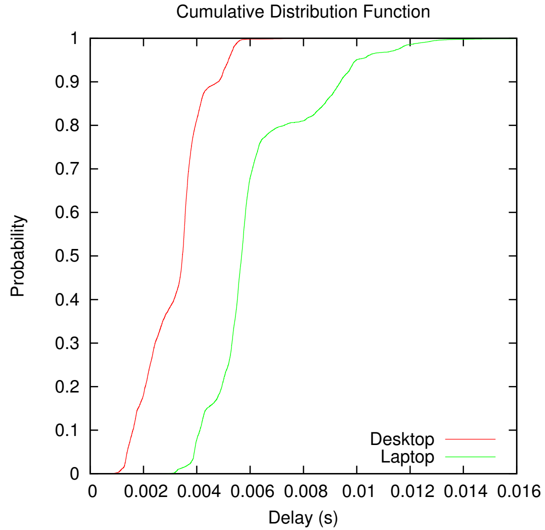

Figure 3.3: Measured network latency

machine is to be simulated, calibration is of course not possible, but then the parameters of the model will have to be defined manually.

Our calibration is done using a calibrator. The calibrator is modelled in statecharts too, and will perform a series of requests. Each request is identical, but the requests are repeated for a certain amount of time, to get a decent amount of samples. The server will be slightly altered, to also time how long it takes from the moment the message is received from the socket, to the time its reply is put on the socket again. This duration is included in the reply as an additional parameter. The calibrator (client) times how long the request took in total, but then subtracts the processing time that it got together with its reply and divides the resulting value by two. Subsequently, the calibrator will have a value for the time it took for the server to process the message, and a value for the time it took the network to transport the message between the client and the server.

These timings are saved and a distribution is generated depending on the measured values. We repeated this calibration for a variety of machines, all of them running both the calibrator and server at the same machine. Calibration results for both parameters are shown for a desktop (*Intel i5-4570 @ 3.2 GHz*) and netbook (*Intel SU3500 @ 1.4 GHz*) in Figures 3.2 and 3.3.

Note that these results are not completely correct, in the sense that the network latency only contributes for a small part to the parameter “network latency”. However, for us, this latency includes the processing of the message in the client, which has the same effect in the end.

3.3.2 Validation

After calibration, there is a need for validation to make sure that the models are indeed correct with respect to our requirements. For validation purposes, we created ’automated clients’, which will perform the desired user behaviour. An MvK server is started and such automated clients are started in a batch file. These values are then compared to a DEVS simulation of the model, with these exact same parameters. Simulation is done using PythonPDEVS [8]. If the model is correct, results should be almost identical.

Results for 10, 50, and 100 clients are shown in Figure 3.4, 3.5, and 3.6 respectively. From these results, it is clear that results are fairly accurate for every amount of clients being validated. For only 10 clients, results are slightly off, though this is to be expected due to the small time scale and small amount of clients.

3.4 Results

After calibration and validation, we have some certainty that our model is correct with respect to the requirements. As per our goals, it is now possible to simulate e.g. the request delay that is to be expected in real life scenario’s, depending on all parameters previously configured. Figure 3.7 shows this, depending on the number of concurrent clients. Using this plot, combined with hard performance requirements, it is possible to estimate the total number of clients that can be concurrently connected to the server. Due to it being a boxplot, requirements can be made about the general case, average case, or worst case.

From this plot, it becomes clear that the complexity of the server is $O(n^2)$ in function of the number of clients. We investigated these results, as this is not a desirable complexity for a server, and found out that this was caused by an inefficiency in our SCCD

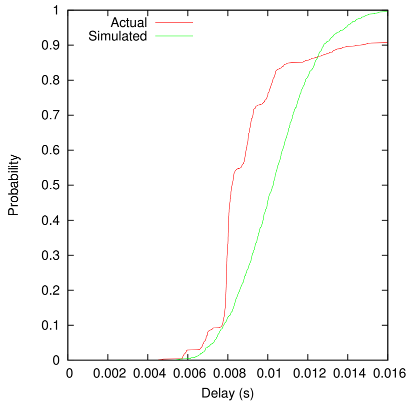

Figure 3.4: 10 clients

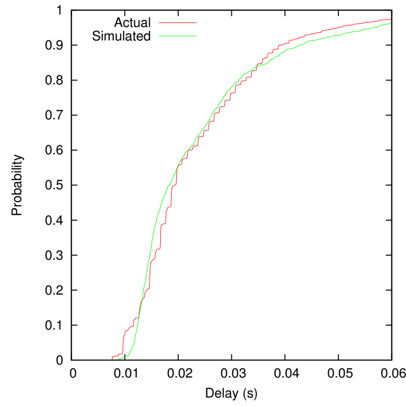

Figure 3.5: 50 clients

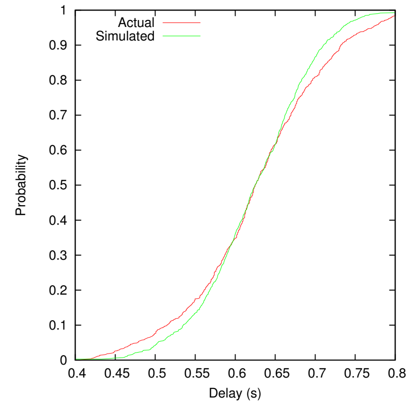

Figure 3.6: 100 clients

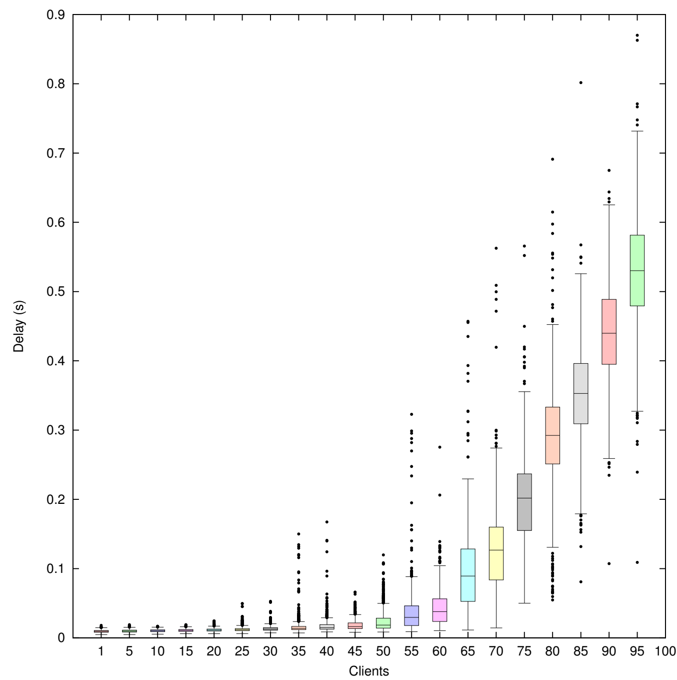

Figure 3.7: Delay for varying number of concurrent clients

compiler and runtime, which does not scale optimally with multiple objects.

Our results learned us that the MvK server performance is highly dependent on the number of clients and the processing time. On the other hand, increasing the network latency has an almost unnoticable effect due to the bad scalability of SCCD.

4

Conclusions and future work

By explicitly modelling the behaviour and performance of the client server architecture of the MvK, we have achieved our initial goals.

- Communication is non-blocking and fully integrated with statecharts. Additionally, due to the modularity, this protocol is again reusable in other projects that use statecharts too.
- Portability is partially achieved by mapping to statecharts. Due to the lack of a complete, language-independent action language, this goal is not achieved completely. Additionally, there is still some small socket-wrapper that is required to make the low-level binding with the sockets in the target language. However, this socket library is much smaller, as it consists of only a few lines of trivial code for every socket operation that is supported.
- Multiple users are supported by using the dynamic creation and destruction of statecharts. No deadlocks or strange interleavings can occur, and the user is completely shielded from all concurrency issues.
- Both low and high levels of abstraction are available to the user, depending on which is desired. The low level of abstraction makes it possible to add additional information in the statechart and the socket library. The high level of abstraction is the one used by the real client and server application, which allows communication with statechart events. As this event passing is even closer to the application being modelled, we believe that this offers an even higher abstraction to the client, whereas it offers the possibility for low abstraction where necessary.
- By explicitly modelling the client and server, up to the level of the sockets, all of the communication protocol can now be analyzed.

Additionally, adding features in the future will become much easier due to the explicit modelling of the reactive behaviour in Statecharts.

By modelling the performance, we gained detailed insight in the complexity of our application, and have even discovered a complexity problem in our statecharts compiler and associated runtime. A calibration tool was created to compute the distribution of both the processing power and the network latency, which can be used to calibrate the DEVS model. After this calibration, validation was performed, which showed that our model closely resembles actual results that are achieved during validation. Finally, simulation was used to determine the influence of the number of concurrently connected clients on the total delay for each request. These results can be interpreted and combined with hard performance requirements, to get a grasp of how many clients can connect to a specific machine.

In the future, we wish to solve the complexity problem of the statechart, to give more favorable performance results for our approach. Also, we could extend our approach and also model our client server architecture in UPPAAL [1], which would allow us to verify properties about our model.

Bibliography

[1] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL - a tool suite for automatic verification of real-time systems. In *Hybrid Systems III*, volume 1066 of *Lecture Notes in Computer Science*, pages 232–243. Springer Berlin Heidelberg, 1996.

[2] Alex Chung Hen Chow and Bernard P. Zeigler. Parallel DEVS: a parallel, hierarchical, modular, modeling formalism. In *Proceedings of the 26th conference on Winter simulation*, pages 716–722, 1994.

[3] Glenn De Jonghe and Hans Vangheluwe. Statecharts and Class Diagram XML - A general-purpose textual modelling formalism. Technical report, University of Antwerp, 2014.

[4] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), jun 1999.

[5] David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[6] Eugene Syriani, Hans Vangheluwe, and Amr Al Mallah. Modelling and simulation-based design of a distributed DEVS simulator. In *Proceedings of the Winter Simulation Conference*, pages 3007–3021, 2011.

[7] S. Van Mierlo, B. Barroca, H. Vangeluwe, E. Syriani, and T. Kühne. Multi-level modelling in the Modelverse. In *Multi-Level Modelling Workshop (MULTI 2014) Proceedings*, 2014.

[8] Yentl Van Tendeloo and Hans Vangheluwe. The modular architecture of the Python(P)DEVS simulation kernel. In *Proceedings of the 2014 Symposium on Theory of Modeling and Simulation - DEVS*, pages 387–392, 2014.

[9] Bernard P. Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of Modeling and Simulation*. Academic Press, second edition, 2000.