Stratmas System Description

Daniel Ahlin Per Alexius Amir Filipovic

January 31, 2007

Parallelldatorcentrum

This document gives an overview of the Stratmas system and some of the main design decisions that have been made during the development process. It describes both the Stratmas client and the Stratmas server as well as the protocol defining the communication between clients and the server, the new Taclan V2 language and the object model used throughout the Stratmas system.

1 The New Description Language - Taclan V2

1.1 Overview

This version of Stratmas introduces a new language for describing the scenarios simulated. The language is basically a syntactic variation of (or layer over) XML Schema, making it extremely versatile and adaptable to further development of Stratmas. The formal grammar of the language is presented in appendix A. There were two primary reasons for choosing this solution:

Type extensibility The models simulated in Stratmas has varied over the years, resulting in several different language versions. The current approach seeks to remedy this situation by having a modifiable type language (XML Schema) and a static syntactic language (Taclan V2). The type language can be altered without needing to modify any parsers or suchlike, it's simply a question of editing a text file (e. g. `StratmasClient/schemas/taclan2sim.xsd`). This means, among other things, that no experience of using for instance parser generators is needed for further development of Stratmas.

XML used between client and server One design constraint given at the start of this project was that the communication between the server and client part of Stratmas should be in readable XML. An implication of this constraint is that an XML-based description (either a DTD or a XML Schema) of the data passed between the client and server side had to be developed to meet this constraint. It was surmised that such a description could, if carefully designed, be used to represent the data not only during transmission, but at any situation. Taclan V2 and the internal data format of the client can be viewed as the result of this conjecture. This will be described in greater detail in Section 2.

It is natural to question why not XML was used all the way. Why introduce a new language which can be expected to be used by only a very limited audience? The primary reason is the anticipated need for a future version having to handle more complex expressions than just declaration of presence of certain objects. Additionally, there is the question of readability of XML (in general), undoubtedly it uses more syntactic markers than strictly necessary for this application. Nevertheless if no complex expressions will be introduced it would probably be a good idea to let the language revert to XML in the future.

1.2 Main Differences From Taclan V1.04

The new language is able to express any scenario expressed by the Taclan V1.04. It should be noted, however, that the expressional power of Taclan V2 is not resident so much in the syntax per se, but rather in the type description used in conjunction with it. In other words, if the XML Schema used is conceptually "near" the data model of Taclan V1.04 (as is the case with the current shipment), then the Taclan V2 file will describe the same kind of scenario. There are however a few notable differences:

1.2.1 Referability

Everything declared (i. e. instantiated) can be referred to, meaning that anything and everything in a Taclan V2 file is uniquely identifiable. This means (among other things) that it is possible to selectively import objects from other files. For instance:

```
import 'Generic platoon' from "Units.tl2" as 'First Platoon'
import 'Generic platoon' from "Units.tl2" as 'Second Platoon'
```

can be used several times in a Taclan V2 file to populate a scenario with units from some standard location.

As can be seen in the second rule for IDs in Section A.2, the language allows defining and using IDs containing (almost) free-form strings, provided they are enclosed in single quotes. The reason for this is to allow users of the language to define meaningful names to the components of the scenario, rather than have to identify them by crammed together names. Compare, for instance 'Red Cross relief team Alpha' with RedCrossReliefTeamAlpha. While the latter is perhaps not unclear to a programmer, to a non-programmer (the perceived target-audience for this language) it may be.

Absolute referentiability in Taclan V2 is based on scoping. Within a scope no two declarations may have the same identifier. The scoping operator `:` is used to delimit a sequence of identifiers describing a path to a specific instance. For instance:

```
import 'Sweden':'Stockholm' from "EuropeanCities.tl2"
```

1.2.2 Strict Language Checking

The implementation of the parser of Stratmas III allows deviance from the Taclan language specification in [2]. This has caused the usage of the language to drift from the specification to some undocumented de facto standard. For instance, the Taclan files made available to PDC contains parts that is not in concordance with the language defined as Taclan Version 1.04 in [2]. A notable example of this is malformed pathnames, pointing out, for instance which ESRI shapefile to use.

In contrast to Taclan Version 1.04, Taclan V2 enforces strict checking from a formal grammar specification, easily over-viewable and verifiable (the syntax and basic semantics are defined in `StratmasClient/TaclanV2/Parser.cup`, and `StratmasClient/TaclanV2/Lexer.lex`).

1.2.3 Non Portable Tokens Used

An interesting and rather exotic feature of Taclan V1.04 is the use of the bullet-symbol (•) as a marker for line comments. While possible to write in many word-processors, type-setting systems, and, indeed, Apple computers. It hinders creation of comments on most other platforms. The keywords of Taclan V2 are few and can be expected to be writable on most platforms in existence today.

1.2.4 Simulator, Optimizer and Language Tightly Integrated

As further discussed below there are implementation, conceptual and most importantly usage-related reasons for splitting simulation, optimization and scenario description. In Taclan Version 1.04 all these components are described and parametrized in the same file.

Via the `import`-feature of Taclan V2 it is easy to split up a simulation description over several different files.

1.3 Deviations from XML Schema in Taclan V2

There are at least two important differences between XML Schema and Taclan V2 (apart from syntactic difference).

Declaration ordering XML Schema enforces strict ordering of sub-elements in a declaration. This is removed in Taclan V2 where the ordering of sub-elements in a declaration is not considered as having meaning.

Modular importation of external resources The Taclan V2 parser employs a modular approach to handling the `import`-statement. Several modules may register with the parser, the first to report success on importing the target is used. (Which makes the following statement possible:
`import 'Sweden' from "Europe.shp" as 'map'`
which will (try to) import the shape identified as 'Sweden' in the ESRI file set `Europe.shp` and `Europe.dbf`).

2 The Data Model

Figure 1 shows a UML-schema describing the main structure of the used data model. This new approach can be divided in three distinct parts:

1. A description of the components of the simulation and scenario. Figure 1 can be viewed as a graphical representation of this part. More mundanely this part is defined by `StratmasClient/schemas/taclan2sim.xsd`
2. A simulation description, described using Taclan V2 or XML.
3. A scenario description, described using Taclan V2 or XML.

The rest of this section contains summaries of these three parts.

2.1 Describing the components from which to build a scenario

A scenario contains objects of many types, several different kinds of units, a large variety of equipment and so forth. This information can be stored in a set of attributes belonging to the object type.

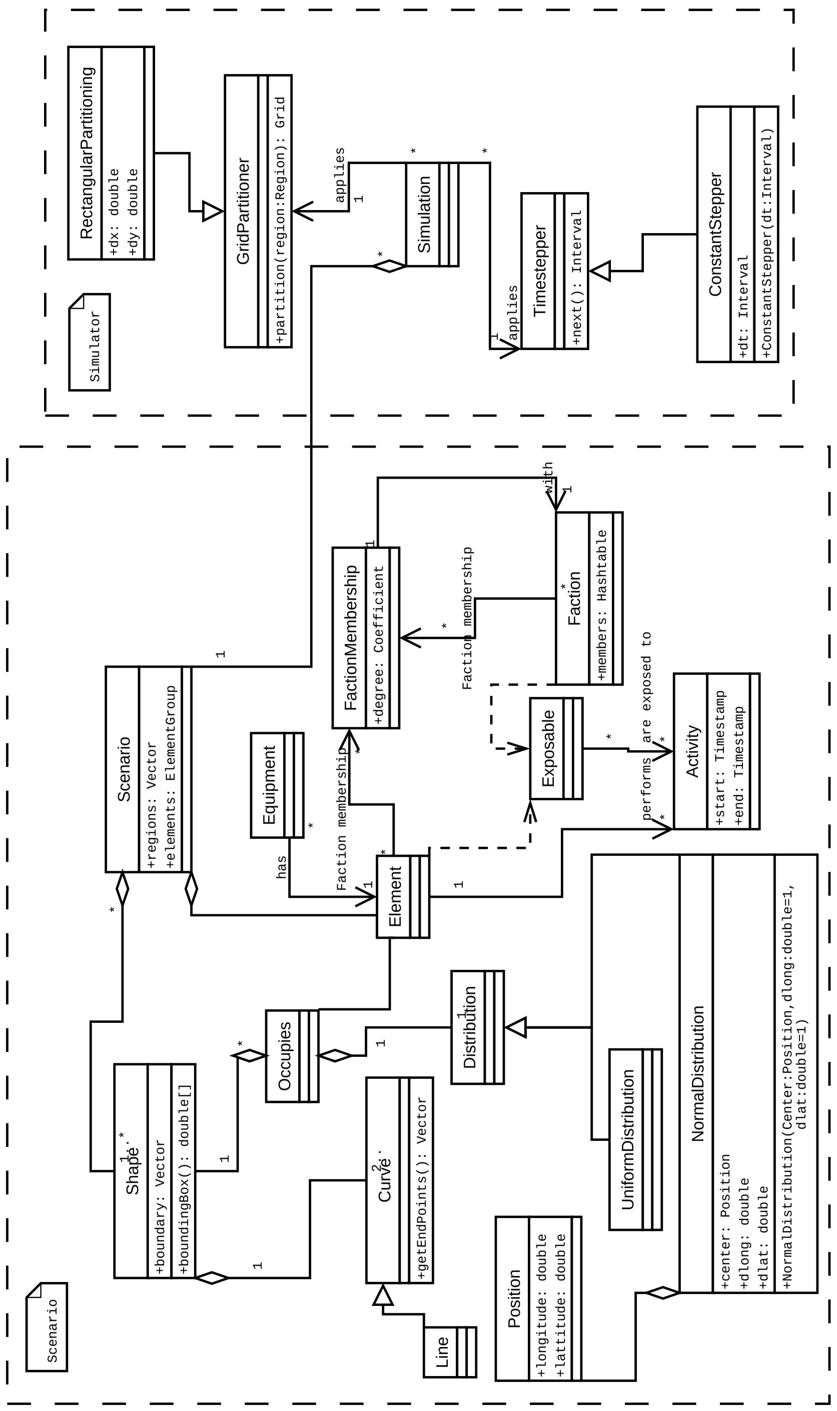

Figure 1. This schema is a schematic view of the data model employed in the Stratmas client. The main deviation from the Taclan V 1.04 is the shift from point location to region based occupancy. It should be noted that previous versions of Stratmas also employed various region-based strategies, however this was not exposed to the user in any great degree. In the new version every detail of the location is available for inspection.

A considerable part of the information describing, for instance, what a `platoon` actually is, will be static over a scenario. Examples of such static information are

- The name of the unit type (i. e. `platoon`).
- The typical action pattern of the unit (i. e. how it will act given some situation).
- The initial conditions of variable attributes, such as `personnel`.
- Most importantly, *what* attributes are used to describe a `platoon`.

The premises of this approach are three important observations regarding this information:

- The people with the best knowledge of how to describe an object in a scenario are not necessarily people who would implement the client or design the scenario language.
- The way of describing an object in a scenario is not constant over time. It can be expected to vary with, for instance, available information, available time and computational resources, and most importantly the requirements and capabilities of employed and available simulators.
- Having complete type definition capabilities in the scenario description language till hamper easy understanding of a scenario description. For many viewers the information that a `platoon` is present is the only information of interest. This category of users should not be confused with information of *how* types are described.

2.1.1 Describing a Scenario

What constitutes a scenario? The view applied in the model is that it consists of observable objects in the world and in the society of its inhabitants. Naturally the model is unable to handle such complexity and limitations has to be imposed. However, the value of the view is that it explicitly separates the description of *what* is simulated from *how* it is simulated.

The main benefit of this is that it effectively limits how restrictions imposed by the state of the simulator (and available computational resources) affects the description of a scenario. The perceived consequences of this are

- Increased lifetime of the scenarios (the simulator may change, the reality of the scenario does not).
- Easy adaptation of the simulation of a scenario to available time and computational resources.
- Easy use of several different implementations (e. g. different generations) of the simulator.

A scenario in the current model is built using components of the following types.

`Activity` is the superclass of the objects that describes processes and actions in the scenario. This is what in Taclan V1.04 where known as Events and Plans. Currently there is only one instantiable subclass of `Activity`, called `CustomPVModification`, which spelled out means custom process variable modification. This class of activities mimics the way plans and events worked in Taclan V1.04, namely by specifying exactly what an event did to the process variables at some location. When a proper combat model has been implemented the orders and rules of engagement is likely to be represented as subclasses of `Activity`.

`Shape` this is the superclass of the objects that define two-dimensional physical surfaces, these are mainly used in conjunction with `Distribution` to describe how an `Element` occupies an area. In the current release there are three different instantiable subclasses of `Shape` — `Polygon`, `Circle` and `Composite`. A `Composite` consists of one or more `Shapes`. Each `Scenario` has exactly one `Shape` that defines the map that is the arena for the scenario.

`Distribution` is the superclass of the objects representing how an area defined by a `Shape` is occupied. The currently instantiable subclasses of `Distribution` are `UniformDistribution` and `StratmasCityDistribution`. The latter is an exponential-like distribution used for population modeling in Stratmas III.

`Element` is the superclass for anything that has a true physical presence in the simulated world. The main attributes of an element are location and deployment, describing where and how the `Element` occupies a certain space on the map. Furthermore an `Element` may have a set of activities tied to itself, defining the actions that it performs. Lastly an `Element` may be affiliated with a certain faction, indicating membership with, for instance a religious faction in society.

Currently instantiable subclasses of this class are:

`MilitaryUnit` is the superclass of all military units, supplementing the attributes in `Element` with, among others, a list of subunits and a symbolIDCode (denoting the App6A symbol of this unit).

`AgencyTeam` is the superclass of the objects representing the agency concept from Stratmas III.

`Population` is a generalization of the city-concept from Stratmas III. Defining a population center, possibly with people belonging to several different factions.

`Faction` is the superclass of the objects that represents societal partitions in the society of the scenario. Currently `EthnicFaction` is the only instantiable subclass of `Faction`, mimicking the Ethnicities-concept in Stratmas III.

`Disease` In each simulation there are exactly one disease with defined rates of infection, recovery and mortality.

Apart from these types there are also primitive types such as integers, timestamps, floating point numbers, dates and so forth, for a more precise definition, consult `StratmasClient/schemas/taclan2sim.xsd`.

2.1.2 Describing a Simulation

The decision to separate the simulation from the scenario description was taken very early in the design process. As stated above, the basis for this decision is, that while the offered options for simulations may vary with available simulation models and available computational resources, a description of a specific scenario should ideally be static.

In current Stratmas there is actually only five variables defining how the simulations are performed on a given scenario, cell size, time step size, random number generator seed, whether to use stochastic or non stochastic simulation, and simulation model. Furthermore, it can be argued that the random number generator seed and the choice of having a stochastic or non-stochastic simulation is actually parameters of the chosen model. Similarly the cell width is a parameter of the chosen grid partitioning strategy. Thus, the general parameters are:

A simulator model The simulation model, e. g. `combat` or `peacekeeping`. The model may have its own specific parameters. At this time the only available server type is `CommonSimulation`.

A grid partitioning strategy The method of dividing the region into cells should be variable. However, currently only square cells are implemented. This strategy is called `SquarePartitioner` and take one variable, the width of the squares, expressed in meters.

A time stepping strategy The simulator will need a method to decide what the next time step is. Currently only constant time steps are allowed. This strategy is called `ConstantStepper`, which takes one variable, the step size in seconds.

3 Stratmas Client

The following sections describes some of the features in the Stratmas client, including the use of Java to achieve multiple platform support and JOGL to enhance graphics rendering.

3.1 Java

The Stratmas client is written in Java which opens the possibility to support multiple platforms. The Stratmas client has currently been tested on Linux, Mac OS X and Windows.

3.2 JOGL

JOGL is a Java programming language binding for the OpenGL 3D graphics API, designed to provide hardware-supported 3D graphics to applications written in Java. It provides full access to the API:s in the OpenGL 2.0 specification as well as nearly all vendor extensions and integrates with the AWT and Swing widget sets.

The main reason why JOGL is used in the development of the Stratmas client is basically its fast rendering of objects by using OpenGL bindings. The

Stratmas simulation is a dynamic process where several objects change their spatial location (or value) several times in a short time period. The number of these objects can be quite high i.e. thousands or even tens of thousands. Each time an object change its location (or value) a map or a part of it has to be redrawn. This frequent redrawing of the map is resource demanding and thus, a fast renderer is necessary. This fast renderer is OpenGL.

Some of the advantages when using OpenGL for graphic rendering are:

- OpenGL provides direct access to the rendering pipeline.
- OpenGL is optimized in hardware and software and targeted platforms ranging from the cheapest PC:s to the most high-end graphics supercomputers.
- Vendors of every kind of 3D graphics-related hardware support OpenGL.

3.3 Implementation Level Tree handling

Based around a data model in the form of a tree, much of the code in the client is concerned with traversing and investigating trees. Two techniques have mainly been used to facilitate this.

Filters First a small set of traversing filters have been implemented, in the spirit (but far from being comparable with) the XML Path Language. This allow shorthand handling of routing tree traversal operations such as collecting all nodes of some specified type. The following can for instance be used to get all `MilitaryUnits` in a simulation:

```
StratmasObject root;
Vector militaryUnits =
  (new TypeFilter(TypeFactory("MilitaryUnit"))).filterTree(root);
```

There are also filters combining other filters with each other and filters acting upon path relations.

Event based Updating To avoid polling tree nodes for update a callback-like approach using `StratmsEvents` has been used. Since a listener is guarantied to receive notifications about removal this also serves as a convenient solution to ensure the non-nullity of critical objects.

3.4 Projection

The projection used in the Stratmas map windows is Lambert's azimuthal equal-area projection. The properties of this projection are:

- It preserves equal area of the projected region.
- It presents true directions from the center of the projection to any other point in the projected region.
- Distortion is zero at the center of the projection and increases radially away from the center point.

By preserving equal area of the projected region, this projection is well suited for displaying spatial data such as military and civilian units, miltary equipment etc. That is the main reason why the azimuthal equal-area projection has been used in the Stratmas client. It should be pointed out that this projection also was used in Stratmas III.

3.5 Color Map

A color map is used for mapping a numerical value to a color value. The color map used in the Stratmas client consists of 256 different colors. It can be created from a whole range of different colors i.e. between 2 and 256 colors. Colors used in the creation of the color map are distributed linearly over the mapping interval and the other colors are computed by using Hermite interpolation with the slope 0. The mapping interval in the color map is adaptive and is defined by the user. The interval can be changed during the simulation which allow the user to specify the interval according to his or her own needs. This allows the user to obtain more detailed information about the different process variables at different time steps, a feature that was not present in Stratmas III where the mapping interval is constant.

3.6 Multiple Map Windows

The Stratmas client allows multiple map windows. This means that several map windows can be opened during the simulation where each map window contains a map over the whole simulated region or just a part of it. Each map window has a separate controlling window.

The multiple map windows allow the user to follow different process variables and/or factions at the same time. Further it is possible to follow the simulation over a whole region or only a part of it. By using this feature, the user can obtain much more information during a simulation than using only one map window which was the only possibilty in Stratmas III.

3.7 Drag'n'Drop

Drag'n'drop (DnD) is an intuitive GUI gesture used for transferring data from one GUI component to another as well as inside a GUI component. In the Stratmas client DnD is used to move objects (military units, civilian units, cities and shapes) from the tree view to the map windows. Some of these objects (military and civilian units) can be moved inside these components as well. This feature allows for easy emplacement of the military and civilian units in the map window. Further on, when used with shapes, it can increase the displayed area in the map window by dropping a new shape. Using DnD in the Stratmas client results in a more friendly user interaction.

3.8 Source Code Documentation

The source code of the Stratmas Client is documented in such a way that it is possible to generate API documentation using Javadoc. The API documentation may then be browsed through using an ordinary web browser. This greatly facilitates the procedure of creating an understanding of the source code for other developers.

4 Stratmas Server

The following sections provides descriptions of various aspects of the Stratmas server, including a brief description of the server and a list of improved simulation capabilities, among other things.

4.1 Brief Description

The Stratmas server is the part of Stratmas that handles the actual simulation calculations. It executes the simulations it gets from a client and sends back the results. The Stratmas server may be initialized and controlled by one client — the so called *active client*. Several clients may be connected to the server simultaneously but all other clients than the active client may only passively watch the ongoing simulation, so called *passive clients*.

4.2 Improved Simulation Capabilities

The simulation capabilities of the Stratmas server is a super set of the capabilities present and working in Stratmas III. The same functionality as provided by the models in Stratmas III is implemented except for the correction of obvious bugs. The parts of the Stratmas III source code that was not used or did not work have not been implemented in the Stratmas server.

Improvements of the simulation capabilities have been made where possible without modifying the original models. Such improvements include:

- Instead of supporting a maximum of only three different ethnic factions as in Stratmas III, the Stratmas server is capable of handling simulations where the number of factions is limited only by the available system resources.
- In Stratmas III the partitioning of the simulation grid did depend on the resolution of the screen, i.e. the same simulation would produce different results depending on which screen that was used to watch the simulation. This anomaly has now been removed which also has the consequence that the actual size of the cells in the grid now falls much closer to the specified size[1]
- Stratmas III limits the number of participating forces to a maximum of three. The Stratmas server supports any number of forces, limited only by the available system resources.
- The Stratmas server allows any number of effects from every activity[2], compared to a maximum of 5 in Stratmas III.

[1] Due to the nature of map projections there will always be some deviation from the specified cell size. See Section 3.4 for more information about map projections.

[2] Called Event or Plan in Stratmas III.

- The area of cities are no longer fixed to circles with a radius of 48 kilometers. The Stratmas server supports cities with areas shaped as circles or polygons of any size. It is also possible to change the form of the function used to distribute the population from a city, an option that was not available in Stratmas III.
- The number of teams in each agency is no longer limited to 25, which was the case in Stratmas III.

4.3 Multiple Platform Support

One of the main design goals for the Stratmas server has been to be able to support multiple platforms. The server currently supports Linux, Mac OS X and Windows. It has also been tested and found working on Solaris although those tests have not been very extensive.

In order to achieve support for multiple platforms from the same code base, the Stratmas server source code has been written to use standard components wherever possible. Large parts of the source code uses nothing outside the C++ standard library. In addition to that, the Boost library has been used in order to reconcile platform differences when it comes to, for instance threads, and the Xerces C++ Parser has been used for XML parsing.

Apart from this, two distinct implementations of network abstraction are currently implemented, one for Windows and one for Posix-like systems (this can be regarded as an interim solution as Boost is expected to provide a unifying abstraction for this area in the near future). Additionally the code concerned with running the server as a daemon or service is by necessity divergent between Posix-like systems and Windows, a difference which unfortunatelly is not expected to disappear.

Stratmas III was a Macintosh only application and did as such make frequent use of Macintosh only system calls. All such calls have been removed and their functionality replaced.

4.4 Main Design Description

This section gives some insight into various design decisions that have been made during the development of the Stratmas server. The main design is described and the reasons for various choices explained.

4.4.1 Data Flow

One of the design criteria for the Stratmas server was to allow multiple clients to connect to the server and watch a simulation simultaneously. In order to achieve this, a design that is schematically described in Figure 2 was chosen. A *Session* is used to store all information the *Server* needs for communicating with a certain client. The *Buffer* is used for storing data extracted from the simulation and the *Engine* transfers data between the *Buffer* and the *Simulation.* Notice that a passive client may only fetch data from the server while the active client also is allowed to send data to the server.

It is reasonable to require that when a client fetches data from the server, all that data should belong to the same simulation time step. It is also plausible to require that clients should be able to fetch data from the latest simulated time step without interrupting the ongoing simulation, that is, to fetch data while the simulation is currently executing. The consequence of the requirements above is that it is necessary to save copies of the data that should be sent to the clients between each simulated time step. These copies – called *data objects* – are stored in the *Buffer*. Each data object except simple types such as integers and floating point numbers has a corresponding *simulation object*, i.e. an object that is used in the actual simulation and in addition to the pure data also contains methods for the object's functionality in the simulation.

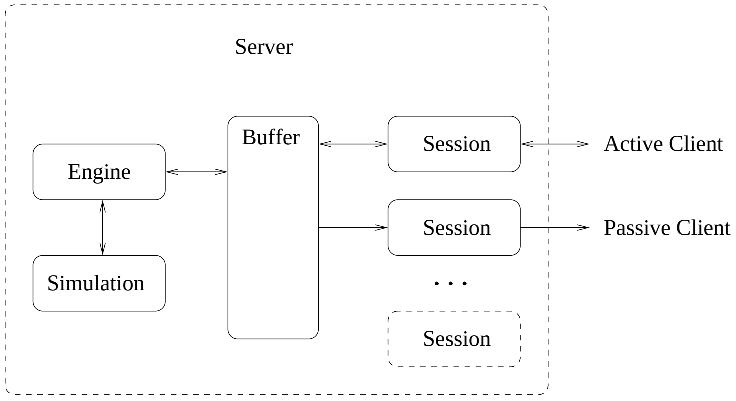

Figure 2. Schematic view of data flow in the Stratmas server. Arrows indicate that data flow in the same direction as the arrow.

4.4.2 Object Model Compliance

The internal structure of the Stratmas server has been built in accordance with the Stratmas object model, described in Section 2. This applies to both the data objects and simulation objects described in Section 4.4.1.

The reason for choosing this design is that it is very convenient from a developer's point of view if the same structure can be recognized throughout all parts of Stratmas. Switching between development of the client and server parts becomes significantly easier and thus less time consuming.

4.4.3 Model Restructuring

The source code that handles the initialization, update and activity exposure of process variables has been restructured in the Stratmas server. The reason for this is that the corresponding source code in Stratmas III was hard to overlook and thus very error prone when introducing changes.

The new structure presents a well defined interface for initializing a process variable, updating it between two consecutive time steps and for exposing it to external influence, for example through activities. Round off errors introduced due to lack of computational precision are also handled through an interface that is the same for all process variables.

4.5 Source Code Documentation

Every class, class member, function and argument in the Stratmas Server source code has been documented. From that documentation an HTML representation has been produced by using Doxygen[3], making it possible to browse through the structure of the source code using an ordinary web browser. This greatly facilitates the procedure of creating an understanding of the source code for other developers.

5 Stratmas Communication Protocol

This section deals with the different aspects of the communication between clients and the server. It is split up into three parts — the infrastructure for sending messages, the way messages are sent and the contents of the messages.

5.1 Infrastructure

Communication between clients and the server goes through stream sockets (i.e. via TCP). POSIX sockets are used on the server side and Java's sockets on the client side.

5.2 Message Sending

Stratmas uses a synchronous scheme for message transfer. It is always the client that initiates communication, i.e. the server will never send anything unless a client has asked for it. This is a simple, straight forward way to handle communication.

The contents of a Stratmas message is XML based. However, in order to be able to send and receive messages in an easy fashion a simple underlying protocol is used. It is schematically described in Figure 3.

Length	Id	XML Content
8 bytes	8 bytes	'Length' number of bytes

Figure 3. The underlying structure of a Stratmas message.

The first field of the message is a 64 bit network byte order integer containing the length in bytes of the XML contents of the message. The second field is a 64 bit network byte order integer containing the server created id of the session and the last field contains the actual XML contents. The id field is currently used as follows. When a client first connects to a server the id field should be set to -1. When receiving the response to that message the id field will be set to a server generated number that is unique for the session just started. The received number should then be used as id for all the messages sent during that session.

[3]See the Development Environment document for more information on Doxygen.

In future releases of Stratmas it should be possible to detach a client from the server, i.e. leave the server working with a simulation while the client is disconnected and shut down. If a client detaches from the server, the received id might be used to reconnect to the same session. This may be useful when simulations take long time or if the client for some other reason may not be running the whole time. The detach functionality is implemented in the Stratmas server but not yet in the Stratmas client.

5.3 Message Content

One of the requirements for the protocol handling the client server communication was that it had to be XML based. With this requirement set, the upcoming technology with XML Schema was chosen. Basicly, this means that two kinds of XML-documents are used to describe the data transfered. The first part is the schema document. This is a document that describes how another XML document should be structured and what kind of contents it should have. It can be thought of as a kind of template for the data. Then there is the so-called instance document that constitutes the actual data that is communicated.

The use of XML Schema gives some rather crucial benefits. Since XML schema allows an object oriented approach and thus supports inheritance, it provides a very close and natural relation to the object oriented design of the applications. The Stratmas object model is represented in the XML Schemas used for client server communication which greatly facilitates the production and consuming of message contents since the objects in the protocol have their natural mapping to objects in the client and server applications.

XML Schema also opens the possibility for an extremely flexible and powerful solution when it comes to the parsing of indata files and the Stratmas client setup. The client is in large part able to visualize whatever contents the indata file contains as long as it validates against the XML Schema. This solution saves a considerable amount of time when introducing new attributes or removing old ones, since the only thing that in many cases has to be done is to update the XML Schema describing the object model. In those cases not a single line of source code has to be changed on the client side.

Another benefit of having the XML representation of objects in the applications built in is that it is a convenient way of exporting such data to other applications for other purposes.

A Formal Definition of the Taclan V2 Language

A.1 Language Keywords and Other Constant Valued Syntactic Tokens

= marks an assignment.

`{ }` marks the beginning and end of a block respectively (e. g. a declaration).

`import` specifies that, at this point, the information stored at some external location should be retrieved and used.

`as` indicates that an result of an `import` should be renamed `from` a pointer to the external location used in the import.

`true` is the marker for the true-value of boolean variables.

`false` is the marker for the false-value of boolean variables.

`:` is the scope operator, used to delimiter identifiers into paths.

`//` marks the beginning of a comment lasting till the end of the line

`/* */` marks the beginning and end of a block comment respectively.

A.2 Syntactic Elements Defined by Regular Expressions

Let ¶ denote the line terminator[4], then the non constant syntactic elements are[5]:

`[A-Za-z_][A-Za-z_0-9]*` is an identifier denoted ID.

`'(((\\)|(\')|(\n))|[^¶\'])*'` is an identifier, denoted ID.

`"(((\\)|(\')|(\n))|[^¶\'])*"` is a string, denoted STRING.

`(-|)[0-9]+` is an integer, denoted INTEGER.

`(-|)[0-9]+"."[0-9]+((([eE]((-|))([0-9]+))|))` is a floating point number, denoted FLOAT.

`(-|)[0-9]+[eE]((-|))([0-9]+)` is a floating point number, denoted FLOAT.

Whitespaces[6] are ignored and any other token is a syntactic error.

A.3 Language Grammar

Let ϵ denote the empty production, then the grammar of the language is:

```
declarations → ε | declaration declarations | assignment declarations |
             → massimport declarations
declaration  → ID ID { declarations } | importation | ID { declarations } |
             → ID = ID { declarations }
massimport   → import STRING
importation  → import reference from STRING |
             → import reference from STRING as ID |
             → ID = import reference from STRING
assignment   → ID = rval
reference    → ID | reference SCOPE reference
rval         → reference | STRING | INTEGER | FLOAT | true | false |
             → { declarations }
```

References

[1] Gnu Grep documentation; http://www.gnu.org/software/grep/doc

[2] Cobb, Loren; Stratmas III Code Documentation; January 2004

[4] Which is encoded differently on different platforms.

[5] See [1] section 5 for an introduction to regular expressions.

[6] What constitutes whitespace differs from platform to platform.