SimCore

June 10, 2005

Contents

1 Introduction

SimCore is a library that allows to write code for discrete event simulations. The usage goals are the following:

- Simulation of nation-wide socio-technological infrastructure
  - 100s of millions of simulated elements (people, computers, etc.)
  - Requires as-of-yet unknown approximation techniques of the elements and processes.
- Obtaining results that are meaningful and provide useful information.
  - Limits approximation mentioned in previous bullet.
- Ease of combining simulations that were developed independently.
  - Simulation of people generating phone calls, and simulation of phone infrastructure. Combining the two allows feedback to session generation in case of e.g. overloaded network.

In a short, SimCore's goal is to provide basis for ceating simulations of various socio-technological aspects, altering their functionality as the used models develop, and combining the simulations into more complex ones. I will use Session Simulator and Network Simulator as examples, because these are the only ones using SimCore at the moment, but the library itself is general and not restricted to communication simulations.

From the usage goals, we have identified several important design goals for the SimCore library, roughly in order of importance:

Extensibility: adding a new functionality is very easy and does not require changing much of the existing code (changes are localized). E.g. adding a new protocol to Network Simulator does not require changing the already existing code (which also means that removing an unused protocol is seamless).

Expressive power: the library does not restrict the user to only certain constructs, the full power of C++ language and any library is available. This is important so that any computation and simulated element interaction technique can be implemented if needed, event if it does not fit well into the SimCore design. Unknown approximation techniques may require unforseen implementation steps.

Conceptual simplicity: so that as it grows, it doesn't become an unpenetrable jungle. The places where the different components interact (like different Network Simulator protocols) should be well defined. This is important so that the simulations can grow without anybody having to know all details about all parts of it.

Scalability: there should not be anything that would result in significantly higher memory usage or lower running speed when using the library, compared to when the simulation code is taylored to a particular application. In particular, it must run efficiently on parallel architectures, and there should not be any inherent bottleneck that would limit the simulation scale based on limited resources (e.g. memory) on a single computing node. It must run on *distributed memory* architectures (better availability, cheaper than shared-memory architectures).

2 Architecture Overview

SimCore is a layer that is responsible for providing the user (an end simulation, such as Network Simulator) with concepts and tools for fulfilling Extensibility and Conceptual simplicity design goals. The Expressive power goal comes for free by SimCore being a library in C++ (as opposed to being a simulation-definition language, which could be limiting in allowed constructs). It is *not* a discrete event simulation engine, e.g. it is *not* directly responsible for passing events between computing nodes, event queue maintenance and synchronization. For this, SimCore uses an external software package. Currently, only DaSSF has been ever tried, but in theory, any parallel distributed memory simulation engine should work. The Scalability design goal is therefore largely determined by the simulation engine used, and SimCore simply tries to impose as little performance overhead as possible. The "library stack" is shown in Figure 1. Note that the end simulation is not supposed to interact with the simulation engine directly (let alone message passing layer), which makes them easily portable (even if a different simulation engine would have to be used on a different computer architecture, the end simulation does not need to change).

2.1 Concepts and Tools Provided by SimCore

There are three major concepts provided (and required to be used) by SimCore:

Entity: primary element of the simulation study (a person, a computer etc.) Element, to which events happen during the simulation, and in which properties we are interested.

Service: an element that determines *behavior* of entities, e.g *how* they respond to events that happen to them.

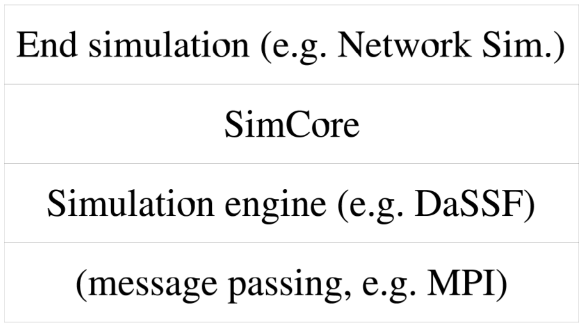
End simulation (e.g. Network Sim.)
SimCore
Simulation engine (e.g. DaSSF)
(message passing, e.g. MPI)

Figure 1: SimCore in an architecture of a simulation.

Info: content of events, e.g. information that is being passed between entities (or, more precisely, services on entities) and which services use to make actions (e.g. sending other infos).

Let's consider the Network Simulator for an example again: the entities would model devices (e.g. computers), interfaces (e.g. network cards) and media (e.g. network cable or a frequency as a wireless medium). Entities, therefore, correspond to *hardware*. Services would model various protocols (e.g. FTP, TCP, IP on a device; RED buffer and CSMA on an interface; and a "signal transmission functionality" on a medium). So services correspond to *software*. Software is installed on hadrware, and it is the interaction of the various software parts that determines overall behavior of the hardware. Infos then model packets that travel on a network, and they are also used to implement passing of various control information (e.g. backoff of a CSMA protocol).

Some most important properties are review bellow:

Entity

- Uniquely identified throughout the simulation system (by an EntityID).
- Has general features (e.g location, capacity, status, etc.)
- Has its own services (each service lives at — is accessed through — a ServiceAddress: known identifier, such as "service that models MAC protocol")
- Forwards Info to an appropriate service (should not handle the info direclty)
- Different entities are *not*, in general, in the same memory space (e.g. on the same computing node). But it is possible to assure that some related Entities are.

Service

- Is NOT uniquely identified (MUST be on some Entity, at a certain ServiceAddress).
- Handles incoming Infos (reacts on them).
- Can only be on ONE Entity.
- Can have multiple ServiceAddresses on the same Entity
- All Services living on the same Entity are in the same memory space (e.g. on the same computing node).

Info

- Data to be exchanged between Entities (or, better, Services on them)
- Info destination address = EntityID + ServiceAddress.
- By default empty, all content is user-defined (in particular, does *not* contain sender's information by default)

An example of a service having multiple ServiceAddresses would be an FTP server accessible via many port numbers. For network applications in general, ServiceAddress can be well thought of as a port number.

Having a clear distinction between objects (entities) and their behavior (services) is the main tool to address the Extensibility and Conceptual simplicity design goals. Adding a new functionality to an entity amounts to adding a new service to it, without having to change existing code. And interaction between different entities is (mostly) restricted to exchanging Info messages (communication between services on the same entity can, nontheless, be made via regular function calls). The concept of services living at ServiceAddresses also allows for easy dynamic change of behavior by simply replacing at run time one service by another at the same ServiceAddress.

3 API

This section describes how are the concepts and tools of SimCore library cast into a C++ code. All of the SimCore code is written in namespace SimCore.

There are some basic types that are user in various places of SimCore:

Time — data type used to represent time. Can be one of float, double, long and longlong (other types can be added to type.h). The choice is made by macro LTIME_FLOAT …, the default is float. There is a constant MINDELAY of type Time defined in constants.h, which specifies the minimum amount of time that events can be scheduled for since the current simulation time (must be greater than zero, a limitation of DaSSF).

Random::TRandom — object that represents a random number generator, can generate numbers from various distributions. See framework/src/Random/Random.h for details.

Ptr¡TYPE¿ — smart pointer to TYPE. Defined in Ptr.h. All pointer operations in SimCore visible to the user are done using smart pointers, so all objects created by SimCore are automatically deleted when no longer needed. This reduces possibility of memory leaks. Ptr¡¿ being 0 means Ptr¡¿ == Ptr¡¿(). The current implementation uses boost::shared_ptr, and has one extension: method `giveup`. This method returns the pointer, and resets itself, therefore transfers the pointer ownership (acting like std::auto_ptr).

Some other types, usually associated with a particular SimCore object, are defined in type.h file.

SimCore executables (executables that use the SimCore library) need to be run with one parameter that specifies a config file to use. In the following description, the content of the config file is referred to.

3.1 Entity, Service, Info

All three main objects in SimCore are represented by a class in C++. User defined objects in an end simulation need to be derived from these base objects (the base objects are not meant to be used directly). This section reviews their basic functionality (public methods and associated functions), while how to derive user objects and the creation process is described in section 4.

3.1.1 Entity

Declared and implemented in files Entity.h and Entity.C. Entities are identified by an **EntityID** (declared in type.h): a 2-tuple of char and long, signifying the type of entity ('p' person, 'd' device, ...), and its number, respectively. The char part serves to easily have EntityIDs unique, especially in cases where two simulaitons are combined (such as Session Simulator and Network Simulator)

Information methods These methods provide basic information about the Entity and the simulation environment.

- `EntityID getId()` — returns an EntityID of the entity
- `Random::TRandom getRandom()` — returns a random number stream
- `Time getNow()` — returns the current simulation time
- `void print(std::ostream&)` — used by `operator<<` for outputting content of an entity (only for debugging purposes)

Info handling methods These methods provide functionality to "intercept" sending and receiving Infos by services. By default they just forward the infos to the simulation engine, or to the service, but can be overriden to do something more.

- `void processOutgoingInfo(const Ptr<const Info>, const Time&, const EntityID&, const ServiceAddress&)` — schedules an Info to be delivered to ServiceAddress at EntityID with Time delay. Ptr¡Info¿ muts not be 0, Time must be >= MINDELAY.
- `void processIncomingInfo(Ptr<Info>, const ServiceAddress&)` — forwards the Info to a service at ServiceAddress to handle (described in section 4.3.3), and outputs a warning if it fails (no service at the address or a service that cannot handle the Info type).

Service handling methods These methods provide access to services living on the Entity.

- `bool getService(ServiceAddress, Ptr<ServiceClass>)` — retrieves a service that lives at a particular ServiceAddress. Returns true iff a service convertible to ServiceClass exists at ServiceAddress ("convertible" means that a pointer to it can be converted into pointer to ServiceClass).
- `void setService(ServiceAddress, Ptr<Service>)` — puts a Service to a ServiceAddress (possibly overwriting the old one), or deletes an existing service at ServiceAddress if Ptr¡¿ is Ptr¡Srevice¿() (NULL).
- `void getServiceAddresses(std::list<ServiceAddress>&)` — returns (in the argument) a list of ServiceAddresses that are used at the Entity (some Service lives there). Does not clear the list first.

Miscellaneous external functions These functions are not methods of Entity, but are defined in the same header file. They provide functionality that can be achieved by using other methods of Entity, but is sufficiently common to be its own function.

- `void getRequiredService(const Entity&, const ServiceAddress&, Ptr<ServiceClass>&)` — just like `getService` method of Entity, but throws an exception if the service cannot be returned (either there is no service at the ServiceAddress, or the service that lives there cannot be converted to ServiceClass). This means that if the function returns, the Ptr¡ServiceClass¿ is *not* 0.
- `void createServices(Entity& ent, const EntityInput::ServiceMap&)` — creates services on an Entity based on input. The creation process is described in more detail in section 4.2.1.

- `int distributeInfo(const Entity& ent, Ptr<InfoType> info, const Service* const exclude = 0)` — lets all services on a entity handle the Info (with a possible exception of "exclude", if that is not 0 - to prevent infinite recursion). This is used when other services may be interested in a certain info (mostly entity status updates). Whether the service can handle the info is decided by trying to convert each service to `InfoRecipient<Infotype>`, for those that can, `receive(Ptr<InfoType>)` is called (see section 4.3.3).

3.1.2 Service

A Service lives on an Entity, at a certain **ServiceAddress** (an integer, defined in type.h). At most one service can live at one ServiceAddress, but the same service can live at multiple ServiceAddresses.

Information methods These methods provide basic information about the Service and the simulation environment.

- `ServiceName getName()` — returns a name of a service. ServiceName is defined in type.h . The meaning of ServiceName is discussed in more detail in section 3.3.2.
- `EntityID getEntityId()` — returns an ID of the entity where the service was created. At creation time, it is assigned to an entity, and this cannot be changed. But its ServiceAddress at the entity can be changed at run time, or the service may not reside at any address at all.
- `Random::TRandom& getRandom()` — the same as in Entity
- `Time getNow()` — the same as in Entity
- `void print(std::ostream&)` — the same as in Entity

Info handling methods These methods provide functionality to send Infos. Receiving Infos is done via inheritance from `InfoRecipient<InfoType>` and method `receive` (see section 4.2).

- `void sendInfo(Ptr<InfoClass>&, const Time& delay, const EntityID&, const ServiceAddress&, const bool invalidate = true)` — sends the info to service living at ServiceAddress at EntityID (possibly itself) with delay (mustbe >= MINDELAY). Ptr¡InfoClass¿ must not be 0, and by default will get invalidated by the call (reset to 0, so that the data it poits to can no longer be changed). A warning is issued if Ptr¡InfoClass¿ is not the only pointer pointing to the Info data (because in that case, the data could be changed after this call using the other pointer, and it is not specified whether or not the change would get sent). If the invalidate parameter is set to "false", then the Ptr¡¿ does not get invalidated, and no warning is issued if multiple pointers to the info exist.

Miscellaneous external functions These functions are not methods of Service, but are defined in the same header file. They provide functionality that can be achieved by using other methods of Service, but is sufficiently common to be its own function.

- `int multiSendInfo(const Service& service, Ptr<InfoClass>& info, const Time& delay, const Container& neighbors, const ServiceAddress& address, const bool invalidate = true)` — same as sendInfo method of Service, but sends the same Info to ServiceAddress on multiple entities (neighbors). Neighbors is an STL container.

3.1.3 Info

Infos can be created in one of two ways:

- Internally, in the user code, e.g in response to receiving some other Info, see section 4.3.2.
- Externally, read from an input file. These Info messages start the whole simulation ("seed" events), and can be also send any time during the simulation.

It is *not* possible for the objects "inside" the simulation to distinguish between these two ways, which can provide a great flexibility in controlling the simulation: artificial events can be injected by hand.

Information methods These methods provide basic information about the Info.

- `Size getSize()` — returns a size of the info. May be overriden, but the default version returns byte size of the structure (sum of byte sizes of data members of the Info-derived structure).
- `void print(std::ostream&)` — the same as in Entity
- `const InfoHandler& getInfoHandler()` — returns an InfoHandler object that can do things with the particular type of the user derived Info (mostly for internal purposes, defined in InfoHandler.h):
  - `create` — construcsts an object of that type, also a duplicating version (using the derived object's copy-constructor). Used in `InfoManager::createInfo` and `InfoManager::duplicateInfo`, see section 3.2.3.
  - `execute` — calls an appropriate `receive` method of a service that is supposed to handle the info. Used in `Entity::processIncomingInfo`.
  - `getClassType` — returns an identifier of the particular info-derived object. Used when packing/unpacking the Info before/after sending it to another machine in EventInfo.
  - `getByteSize` — returns byte size of the objects, used by the default `Info::getSize`.

Packing and Unpacking Because SimCore works in *distributed memory* environment, it is sometimes necessary to exhange information (in form of Infos) between different computing nodes using a network connection. To do that, the info needs to be put into a sequence of bytes (packing) that can be later reinterpreted as the original Info (unpacking).

Datastucture called `PackedData` is used as an intermediate format (coded by packing and decoded by unpacking). The structure is defined in PackedData.h and provides overloaded `add` and `get` methods for all basic datatypes (including most common STL containers).

- `void pack(PackedData&) const` — called whenever the Info needs to be send to another computer, anytime between calling Service::sendInfo (or equivalent) and the time when the Info must be delivered. It is *not* specified whether or not it will get called if the Info's recipient resides on the same computing node (it will not, for efficiency reasons, but should not be relied on).
- `void unpack(PackedData&)` — called at the recipient's side, whenever `pack` was used on the sender's. The user-derived Info object is first created using the default constructor, and then this method is called to fill its data from the PackedData format.

Reasonable defaults are provided for both methods. They interpret the content of the object as byte sequence, and send it over as-is. This means, that this will work *only* for Plain Old Datatypes (char, int, long, float... NOT pointers, references and STL containers, but it WILL work for boost::tuples, so EntityID is OK).

Input For Infos that the user wishes to be input-able for a file, it is necessary to specify how to fill in the Info's content from an input record. The input scheme details are described in section 3.3, so the used methods are only stated here. For Infos that the user only wants to create internally, the following methods can be left unoverridden.

- `void readData(Input::DataSource&)`
- `void readProfile(Input::ProfileSource&)`

3.2 Managers

Each of the three basic objects has a manager, defined in EventManager.h, ServiceManager.h and InfoManager.h. The managers are responsible for creating and maintaining the respective objects. All managers are acessible through singletons: `theEntityManager()`, `theServiceManager()` and `theInfoManager()`, so they can easily be used anywhere in the code. No SimCore object is created directly using operator `new`, rather, they are created via calls to methods of the managers. They always return a `Ptr<>` smart pointer, so no dealocation is necessary (no calls to `delete` operator).

Every user-defined C++ object derived from one of the SimCore's three basic objects needs to be registered, and this is also one using the managers. The registration is necessary because, for example, the system needs to know what C++ object to create when a particular identifier (number, string,...) is found in input.

3.2.1 EntityManager

EntityManager is responsible for creating entities, and letting user code access them later, if needed. In the current model, Entities are created on Logical Processes (LPs), and there might be several LPs in a single unix-process[1]. LPs are identified by LPIDs. Before creating an Entity, it is necessary to decide which LP it will live at. Entitymanager is defined in EntityManager.h.

Entity information Provide information about Entities in the simulation system.

- `LPID findEntityLpId( const EntityID& id )` — returns an LPID where the given entity lives at (no matter whether on the current unix-process or a remote one). This function will return without any warning even if an EntityID of an nonexistent entity is returned! See section 4.1.1 for details.
- `bool getEntity( const EntityID& id, Ptr<EntityClass>& )` — returns (in its second argument) a pointer to an Entity, given its ID. Returns true iff the entity if found and lives in the current unix-process. So a return value of "false" means either that the Entity does not live in the current unix-process (the one from which the method was called), that it does not exist at all, or that the entity with the id lives in the current unix-process, but cannot be converted to type EntityClass. If the last case occurs, a warning is issued.

Entity creation There is only one method that creates Entities, all from an input filed, and called before the simulation starts — `void createEntities(const Control::LpPtrMap& lps, const std::string& dataFile)`. Support for creating individual entites while the simulation is running would have to be added.

Registering

- `template<class EntityClass, class InputClass> void registerEntity( const Entity::ClassType& id, void (*preCreator)( const EntityID&, const InputClass&) = 0)` — this registers an object EntityClass derived from Entity which uses InputClass as input parameters in constructor with input identifier id, and, possibly, a preCreator function which will be called for each Entity input record prior to creation. See section 4.1.1 for details.

[1]It is not clear whether the LP concept is necessary at all

- `void registerPlacingFunction( bool (*)(const EntityID&, LPID&) )` — register a function that will be used to determine which LP will an Entity live on. The functions will be called in the order they were registered, until a first one that return true is found. In that case, the LPID returned in the second parameter is used. If none returs true (or none are registered), a default function is used (Entity number modulo number of LPs). See section 4.1.1 for details.

3.2.2 ServiceManager

Defined in ServiceManager.h.

Service creation

- `void prepareServices(const std::string& protFile)` — prepares services (does not create any) to be later created on entities. Basically, read in service input file(s) and stores the content. Must be run prior to creating any services.
- `Ptr<Service> createService(const ServiceName&, Entity&)` — creates a service ServiceName (prepared by `prepareServices` on Entity. This member function is called from Entitye's miscellaneous function `createServices`, but can also be called "by hand".

Registering Services are registered using method `template<class ServiceClass, typename EntityClass, typename InputClass> void registerService(const Service::ClassType& servClass)`, which registers a C++ object of type ServiceClass derived from Service, which is intended to live on entity of type EntityClass and uses input (constructor parameter) of InputClass. The identification of the C++ object is servClass. See section 4.2.1 for details.

3.2.3 InfoManager

Defined in InfoManager.h.

Info information Method `template<class InfoClass> const InfoHandler& getInfoHandler()` returns an appropriate InfoHandler (see section 3.1.3) for the Info type supplied as the template argument. Is used mostly for internal purposes.

Info creation Internally (not from a file), an Info can be created in one of three ways: if its *numerical* (of type Info::ClassType) identification is known; if its *C++* identification (it's C++ type) is known; or making a duplicate copy of an already existing Info. See section 4.3.1 for details about creating Infos.

- `void createInfo(Info::ClassType type, Ptr<Info>&)` — creates an Info if its identification type is known. Uses default constructor of the Info type to be created.
- `template<class InfoClass> void createInfo(Ptr<InfoClass>&)` — creates an Info is its C++ identification is known. Uses default constaructor of the InfoClass.
- `template<class InfoClass> void duplicateInfo(const Ptr<InfoClass> orig, Ptr<InfoClass>& copy)` — makes a duplicate of an already existing Info. Uses copy constructor of the InfoClass. Variations with "const" pointers are also defined.

Methods `void prepareDataFiles(const Control::LpPtrMap& lps, const std::string& infoFiles)` and `void readDataFile(int fileId)` are used for creating Infos externally (from a file). See section 3.3.3 for details.

Registering Method `template<class InfoClass> void registerInfo( const Info::ClassType& classType = Info::ClassType() )` is used to register user objects InfoClass derived from Info. The classType parameter specifies its numerical identification, and must only be specified for Infos, which the user wished to input extenrally (from a file) and must be a positive number. If left unspecified, the system assigns a numerical identification automatically, and the user can still create and use such Infos by creating them using their C++ identification.

3.3 Input

Input correspond to the three basic objects: Entity, Service and Info input. Each of the inputs is in separate file(s), and multiple files can be specified for one input type. The config keys used are ENTITY_FILES, SERVICE_FILES, and INFO_FILES, respectively (multiple filenames are separated by a whitespace on the same line). All files start with a header line, which begins with #.

Each line corresponds to one input record (e.g. one Entity to be created) and is of a variable length (different entity types will require different number of inputs). Each record means that some object in the simulation (Entity, Service or Info) will be created. First few fields in a record have fixed meaning for all records, the rest has user-defined meaning and may have varying number of fields (including none)

Because many input settings may be the same for many objects, an complete input item (data that will be given as a parameter to a constructor of the object being created) is devided into two parts:

Profile: values that are shared by many objects. They are only read-in once. They are specified in the main config file, or included in it using `include "profileFile"` directive, and referred to from the input files.

A profile is an entry of the form:

```
set EntityProfile <ProfileNumber> active { }
```

(the EntityProfile becomes ServiceProfile or InfoProfile as needed, and ProfileNumber is a positive integer, unique for each *Profile family) Inside the {} are `KEY <whitespace> VALUE` pairs (only one on a line, VALUE extends to the end of the line) that represent the actual input values.

Custom Data: this is input data that is unique to the object being input. This is exactly the length-varying part of the input record with user-define meaning.

The fixed-meaning fields in the input record vary between Entity, Service and Info input. But the last of these fields is an integer specifying the Profile to use for that input item. Therefore, this profile id plus the rest of the input record constitute information that will be used to fill in an input item. This item is then given to a constructor of Entity, Service of Info.

An input item is a structure derived from class Input (defined in Input.h). It contains user-defined data items that will used in the constructor (they will mostly be publicly available data members, because input item is nothing more than a data holder). It is responsible for parsing content of the inputs (both profile and custom data) from its textual representation into the proper data format. For this, the user needs to override the following two methods:

- `void readData(DataSource&)` — for reading in custom data part. The DataSource structure is a std::istream containing the rest of the input record from the input file. User can pars it using operator¿¿ . After this method is called, a status of the DataSource is checked and a warning issued if it is failed.

- `void readProfile(ProfileSource&)` — for filling in the profile part. ProfileSource is a ProfileHolder structure defined in ProfileHolder.h, which provides `template<class Result> bool get(const KeyType& key, Result& result)` method for automatic parsing of the `KEY VALUE` pairs into the desired type (operator¿¿ of Result type is used for the conversion from VALUE). It returns true iff the KEY was found in the profile and the conversion was successful.

Any of the above methods can be left unoverloaded, in case that the praticular input part is not needed. But if it is overloaded, it should call the corresponding method of the parent object.

The `readProfile` method is called only once for each profile id (in each *Profile family) on each computing node. Each time a ProfileID is encountered in an input record, it is looked up. It if was never used before, a new user-defined Input-derived object is created (based on what input class was registered with the object being created, see section 3.2). Its `readProfile` method is called and the object is stored for later use. If it was already used, the stored object is recalled. Then, a copy of the input object (using its copy constructor) is made, on which the `readData` method is called. The result is then used as a parameter to a (Entity,Service or Info) constructor (and deleted afterwards). This way, the content of the "original" input object with its profile part filled in is always kept. This means that references and pointers to its data members are valid throughout the simulation.

Input object may also have the `void print(std::ostream&) const` method overloaded for debug printing.

3.3.1 Entity Input

Has the following format:
`EntityId ClassType ProfileId CustomData....`
where the EntityId is the ID of an entity to be created, ClassType is its associated C++ structure (the ClassType is given at registration time, see section 3.2), ProfileId specifies the EntityProfile to use.

Inputs for entities should be derived form EntityInput class (defined in Entity.h). Its `readProfile` method allows for automatic parsing of SERVICES profile entry. The value corresponding to the SERVICES key is a sequence of ServiceAddress=ServiceName entries (separated by a whitespace), which specifies which services should be created on the entity at which address (see section 3.3.2 for meaning of ServiceName). This way, a data member `EntityInput::fServiceMap` is filled in, and services on an entity can be created using a call to `createServices` function in the entity's constructor (see section 3.1.1).

The Entity input file is read before the simulation begins in call to `Control::createEntities()` function.

3.3.2 Service Input

Has the following format:
`ServiceName ClassType ProfileId CustomData....`
where the **ServiceName** (std::string) is an identification of the pair (ClassType,Input item). The ClassType is the service's associated C++ structure (the ClassType is given at registration time, see section 3.2). The Input item is the combinatin of the profile (identified by the ProfileId) and CustomData parts. Therefore, ServiceName can be looked upon as an identifier of a "program" (the C++ class) plus its "config files" (the input item).

The Service input file is read before Entities are created by a call to `Control::prepareServices()` functoin.

3.3.3 Info Input

Has the following format:
`Time EntityId ServiceAddress ClassType ProfileId CustomData....`
Info itself is derived from Input, so it has its `readProfile` and `readData` methods to fill itself with data form a file (using the ProfileId and CustomData fields). The other fields specify that the user-defined Info-derived object with numerical identifier ClassType (see section 3.2.3) will be delivered to service at ServiceAddress on EntityId at time Time.

The Info input file is the only input file that is time-dependent. It is read during the while simulation (not the whole file at once, Infos are scheduled as their time of delivery draws near).

3.4 Output

Output is done through functions in namespace `Output`. Only one output file is created, specified with config file key OUTPUT_FILE. It is record-oriented, i.e. each output operation results in one record (line) in the output file. Each output record has an associated type (an integer), and different record types may be of different leghts (in fact, it is only up to the user to make all output records of some type "semantically compatible").

The output is done via call to functions `Output::output(const Entity&, const OutputRecordType)` or `Output::output(const Service&, const OutputRecordType)` (they only differ in whether they take Entity or Service as a parameter that specifies where is the output coming from). They return an output stream, into which any data (whatever user wants to output) can be send with using `operator<<`. A separator (tab) is automatically added between each item being output. Next time the `output` function is called, a newline character is appended, therefore closing the previous record.

The fields in each output record are:
`Time EntityId ServiceName Type CustomData....`
Time is the simulation time when the corresponding `output` was called, EntityId is the entity from which it was called, ServiceName is the name of the service which called it (or "0" if called from Entity) and Type if the output record type specified. Following are custom data items that are sent to the returned output stream using `operator<<`.

3.5 Miscellaneous

3.5.1 namespace Control

This namespace provides basic functionality of the simulation.

Main functions These functions should be called in the simulation `main()` function, in this order:

- `void init(const std::string& modulename)`
- `void prepareOutput()`
- `void prepareServices()`
- `void createEntities()`
- `void startSimulation()` — this method does not returns until the simulation is over.

The following functions provide basic information about the simulation environment:

- `int getNumMachines()` — returns the number of computing nodes used.
- `int getRank()` — returns rank of current process (0..getNumMachines()-1)
- `getNumLPs()` — returns the number of LPs (may be higher than number of machines).

The Control namespace makes use of the following config settings: SEED (for random seed, the rank of a process gets added to this so that each of the parallel processes have different random seeds), NUMBER_LPS (how many Logical Processes to use, uses LPID modulo rank to decide which process will have which LP), END_TIME (determine till when will the simulation run. It always starts at time 0).

3.5.2 namespace Logger

Provides logging facility. The functions `error`, `warn`, `info`, `debug1`, `debug2`, `debug3` all return std::ostream& wher the log data is send to. They are listed in the order of decreasing urgency. Input settings LOG_COUT_LEVEL, LOG_LEVEL, LOG_FILE and LOG_ABORT_ACTION determine control the logging facility.

3.6 Simulation `main()`

The end simulation `main()` (usually `void ModuleMain()` in our framework) will look like this:

```
void ModuleMain()
{
    /// reads in main config file
    Control::init("SessionSim");

    /// registers all user-defined SimCore-derived objects
    SessionSim::registerAll();

    /// prepares output mechanism
    Control::prepareOutput();

    /// loads services from a file for later creation
    Control::prepareServices();

    /// creates entities (and services on them) from input file
    Control::createEntities();

    /// starts the simulation (read the Info input file(s))
    /// does not return until done
    Control::startSimulation();
}
```

The order of the calls is important.

4 Working with User Defined Objects

This section describes basic steps in usign the SimCore API to construct an end simulation. Each end simulation is written in a separate namespace (e.g. namespace SessionSim), but for ease of using the SimCore library (which is written in namespace SimCore), `using namespace SimCore` directive is used.

4.1 Working with Entities

Each entity in the end simulation *must* be derived from Entity class (defined in Entity.h), or Entity's descendant, e.g.:

```
class MyEntity : public Entity
{
    public:
        MyEntity(const EntityID entId, LP& lp, const MyEntityInput& input);
        virtual void print(std::ostream&) const;

        int myMethod();
    private:
        int fMyProfileItem;
        int fMyCustomDataItem;
};
```

Each derived entity must have a constructor with the following parameters:

- entId — EntityID assigned to this entity (in Entity input file)
- lp — Logical Process where this entity will live (only to be passed to SimCore::Entity constructor)
- input — Input structure derived from SimCore::EntityInput, containing the entity-specific input (as described in section 3.3)

SimCore::Entity's constructor uses exactly the same parameters, so they are just passed. At the end of the constructor, services that will be living on the entity should be created, which can be done by a call to `createServices` function (see section 3.1.1). Each derived entity should also override the `print` method, where the user-defined data (`fMy*Item`) is printed (this only serves debugging purposes). In addition, the derived Entity may define methods of it own (e.g. `myMethod`).

Also, methods `processIncomingInfo` and `processOutgoingInfo` can be overriden if the default behavior (described in section 3.1.1) is not appropriate. They can be used to ”spy” on all incoming and outgoing infos from all services at the entity.

The MyEntityInput might look like:

```
struct MyEntityInput : public EntityInput
{
    virtual void readData(DataSource& ds)
    {
        EntityInput::readData( ds );
        ds >> fMyCustomDataItem;
    }

    virtual void readProfile(ProfileSource& ps)
    {
        EntityInput::readProfile( ps );
        if( !ps.get("ITEM", fMyProfileItem) )
        {
            Logger::error() << "MyEntityInput cannot find ITEM in profile" << endl;
        }
    }

    virtual void print(std::ostream& os) const
    {
        os << fMyProfileItem << "," << fMyCustomDataItem;
    }

    int fMyProfileItem;
    int fMyCustomDataItem;
};
```

(note the calls to EntityInput’s readData and readProfile methods) The EntityInput structure itself has a container (`fServiceMap`) that contais information about which services should be created on the Entity (and is given as parameter to the `createServices` function). This container is filled from the ProfileSource structure (key SERVICES, see section 3.3.1) in `EntityInput::readProfile` method.

Before a user-defined entity can be used, it needs to be registered. This is done using a method of EntityManager (see section 3.2.1):

```
theEntityManager().registerEntity<MyEntity, MyEntityInput>("myentity");
```

This call is usually made from register.C file in the end simulation code, but can be done anywhere *before* call to `Control::createEntities()` The string ”myentity” is the ClassType of the C++ MyEntity object, and is used in the Entity input file (see section 3.3.1).

4.1.1 Creating Entities

All entities are created automatically from the Entity input file during the call to `Control::createEntities()` function (see section 3.5.1) just before the simulation starts.

The *whole* Entity input file(s) is read on *all* simulating nodes (all machines). If the `preCreator` argument is specified in the call to `EntityManager::registerEntity` function of the appropriate Entity ClassType (see section 3.2.1), then the specified function (preCreator) is called for *every* record in the input file (no matter whether or not the entity will be created on the machine). This is useful if some input-specific operations are needed for every input record (like counting number of people in the simulation).

Then, a placement function is called. This function is responsible for deciding which LP will the entity be created on, based *only* on the EntityID. The same function is later used to find which LP each entity lives on. The default placement function is "Entity number (second part of its ID) modulo number of LPs". User may use any function and register it with a call to `EntityManager::registerPlacingFunction( bool (*)(const EntityID&, LPID&) )`. Such functions take an EntityID (as their first argument) and output the place for it (in their second argument) and return true, or return false if they cannot decide the LPID. The Entity's placement is then decided by calling the user-registered functions in the order they were registered, until some return "true". If none of the user-registered functions return true, then the default function is used. So each user-registered function may be capable of placing only one "type" of Entity (decided using the first element of the EntityID), and returns false for all other entities. It is important to note that the placement function must return location *consistently*, e.g. it must not change its result during the simulation[2].

Use of the placementFunctions allows to have constant-time entity-location look-up (asuming there is a "constant" number of placing functions) when we only know the entity's ID (which is the case when we send messages). Using a map from EntityID to LPID would not be constant, and it would also impose a limit on how many entities can be handled by the overall simulation (the map could not exceed memory limits of one process). Using placement functions does not impose such limits. Moreover, the map-usage approach can be used within the placement function one: the `preCreator` function would determine the entity's location (based, for example, on a value from the CustomData input part) and store it in a "global" map, where placement function could later find it.

4.1.2 Accessing Entities

In most cases, the user does not have to access entities explicitly. Every time an info is received, it is delivered to the recipient entity and service automatically

[2]Unless an Entity migration scheme is implemented, which would allow for synamic load balancing.

(see section 4.3.3).

But, if need be, entities can be accessed via a call to `EntityManager::getEntity` (see section 3.2.1). But only entities living on the same simulating node (possibly different LPs) can be accessed in this way.

4.2 Working with Services

Every user-defined service *must* be derived from class Service (defined in Service.h), or Service's descendant. In addition, it must also be derived from template class `InfoRecipient<InfoType>` (defined in InfoRecipient.h) for *each* `InfoType` info that it can receive. This results in multiple inheritance, but inheritance from `InfoRecipient` is purely an interface inheritance. The `InfoRecipient` class has only one pure virtual method `void receive(Ptr<InfoType>)`. This method is the one that will get called whenever an Info of type `InfoType` is delivered to the service.

```
class MyService :   public Service,
                    public InfoRecipient<MyInfo>
{
    public:
        MyService(const ServiceName& name, MyEntity& ent,
            const MyServiceInput& input);

        virtual void receive(Ptr<MyInfo>);

        virtual void print(std::ostream&);

        void myMethod();
    private:
        int fMyItem;
};
```

Each derived service must have a constructor with the following parameters:

- name — name of the service (from Service input file)
- ent — Entity that the service lives on, assigned at registration time (see bellow). A Service may have multiple constructors of this form with different Entity to love on (in which case it can be created on more-than-one entity types, but must also be registered multiple times)
- input — Input structure derived from SimCore::ServiceInput, containing the service-specific input (as described in section 3.3)

SimCore::Service's constructor uses exactly the same parameters, so they are just passed. Every `receive` method inherited from all `InfoRecipints` must be declared and implemented. Each derived service should also override the `print` method, where the user-defined data (`fMyItem`) is printed (this only serves debugging purposes). In addition, the derived Service may define methods of it own (e.g. `myMethod`).

Working with MyServiceInput is the same as with MyEntityInput (see section 4.1). The only diference is that service inputs are derived from `ServiceInput` (defined in Service.h)

Before a user-defined service can be used, it needs to be registered. This is done using a method of ServiceManager (see section 3.2.2):

```
theServiceManager().registerService<MyService, MyEntity,
    MyServiceInput>("myservice");
```

This call is usually made from register.C file in the end simulation code, but can be done anywhere *before* call to `Control::prepareServices()` The string "myservice" is the ClassType of the MyService C++ object, and is used in the Service input file (see section 3.3.2).

4.2.1 Creating Services

Services are created on-demand, which means that user need to specify when and where to create it. It is usually done at the end of entity constructor and via call to `createServices` function (see section 3.1.1), which creates and places (to ServiceAddress) services specified in the entity's profile input with key SERVICES. But it can also be done any other time (after the entity where the service will live is created, of course) and "by hand" via call to `ServiceManager::createService`:

```
Ptr<Service> myService =
    theServiceManager().createService("myservice", myEntity);
```

This way, the service is created to live on myEntity, but it is *not* assigned to any ServiceAddress yet. To do so, call:

```
myEntity.setService(<ServiceAddress>, myService );
```

4.2.2 Accessing Services

Services on the same entity can easily cooperate via calling each others methods directly. This approach is used whenever no progression of simulated time is desired or needed. Sending Infos (to itself or another service on the same entity) is used when progress in time *is* required.

Direct access to a service is provided by method `getService` of an entity where the requested service lives, or by function `getRequiredService` (see section 3.1.1). They only differ in whether or not it is an error if the requested service cannot be returned. The C++ type of the service that we wish to retrieve *must* be known a-priory (it is specified in form of type of the pointer to contain the return value). Then, any public methods of the returned service can be called, including `receive` methods.

4.3 Working with Infos

Every user-defined info *must* be derived from class Info (defined in Info.h), or Info's descendant. Often, an info can be viewed as a data container, so it is reasonable to make its data members public.

```
struct MyInfo : public Info
{
    virtual void print(std::ostream&);

    int fMyInfoItem;
};
```

Each derived info must have a default constructor, and should have a *deep* copy constructor if the default is not (e.g. if it has pointers, it should copy the content of the pointer, not only the pointer itself). Each derived info should also override the `print` method, where the user-defined data (`fMyInfoItem`) is printed (this only serves debugging purposes). It is perfectly reasonable to have derived infos which have no specific data members (and therefore don't need to override any methods) to model various events.

If the info contains more than only Plain Old Datatypes (POD, e.g. int, float, ...), then it *must* overload `pack` and `unpack` methods (see section 3.1.3). Overloading these methods is also necessary whenever the simulation is to be run in heterogeneous environment, where sizes of PODs can vary.

If the info is to be read in from the Info input file(s), then it is necessary to overload `readData` and `readProfile` methods (see section 3.3.3).

Before a user-defined info can be used, it needs to be registered. This is done using a method of InfoManager (see section 3.2.3):

```
const Info::ClassType myInfoType = 4;
theInfoManager().registerInfo<MyInfo>( myInfoType );
```

This call is usually made from register.C file in the end simulation code, but can be done anywhere *before* call to `Control::startSimulation()` The constant `myInfoType` is the ClassType of the MyInfo C++ object, and is used in the Info input file (see section 3.3.3). It is only necessary to specify this parameter if user wishes to input the info from the input file (externally), i.e. whenever methods `readData` and `readProfile` are overloaded. The ClassTypes must be unique across different infos.

4.3.1 Creating Infos

Infos can be either crated on demand, whenever user neds them (internally), or automatically from an Info input file (externally). The recipient cannot find out whether the recieved info was created internally or externally.

Internal creation Used mostly in methods of services, as a way of exchanging information. They are created by a call to `InfoManager::createInfo` for empty infos, or `InfoManager::dusplicateInfo` for copy of already existing infos. See section 3.2.3 for details about the methods, and section 4.3.2 for an example.

External creation Infos sent from the Info input file(s). They are created automatically and are delivered to specified service at specified time (specified in the input file, see section 3.3.3).

4.3.2 Sending Infos

This sections only concers infos that are created internally (see section 4.3.1), because externally created infos are sent automatically.

After an info is created, they are filled with data, and then sent off:

```
/// create it
Ptr<MyInfo> myInfo;
theInfoManager().createInfo( myInfo );

/// fill it with data
myInfo->fMyInfoItem = 1;

/// send it
sendInfo(myInfo, delay, destDevice, destServiceAddress);
```

The `sendInfo` is a method of Service, where most infos will be sent from. By default, it invalidates the info pointer (`myInfo`) so that the user cannot modify the content of the info after the call (it is unspecified whether or not the change would be delivered!) and issues a warning if another pointer to the same info exists (because that could still be used to change it). An optional fifth parameter (`invalidate`, see section 3.1.2) can be set to ”false”, in which case the pointer parameter will not get invalidated, and no warning will be issued if multiple pointers exist. It is a responsibility of the user not to modify the info after the call to `sendInfo` in such situation. The `delay` parameter must be greater or equal to constant MINDELAY defined in constants.h, or else a warning is issued and the info is delivered later (with delay equal to MINDELAY). This implies that infos *cannot* be delivered immediately (a common restriction imposed by many discrete event simulation engines).

The Service's `sendInfo` method calls `processOutgoingInfo` method of an entity where the service lives (see section 3.1.1). The default version of `processOutgoingInfo` calls appropriate functions of the simulation engine to dispatch the Info (see Entity.C for details), but can be overriden.

4.3.3 Receiving Infos

The simultion engine first handles an incoming Info to a `processIncomingInfo` method (see section 3.1.1) of an entity that is a recipient of the info (infos are addressed to (EntityID, ServiceAddress) pairs). The `processIncomingInfo` method is guaranteed to receive the only pointer to the info (so the method can do anything it wants with the info). The default version forwards the info to the repient Service, by a call to `InfoHandler::execute` (see section 3.1.3 and file Entity.C). If the recipient service does not exist, i.e. there is not service at destination ServiceAddress on the entity, a warning is issued and the info dropped. If the service at destination ServiceAddress cannot handle the given Info type, i.e. it does not have an appropriate `receive(Ptr<InfoType>)` method, a warning is issued also and the info dropped. The `processIncomingInfo` method can be overriden.

So, by default (and if all goes well), the info is finally delivered to the destination service by a call to the service's `receive(Ptr<InfoType>)` method, where `InfoType` is the C++ type of the info that was sent.

5 SimCore Internals