Integrated Simulation of

Communication Networks and Logistical Networks

Using Object Oriented Programming Language Features

to Enhance Modelling

Markus Becker, Bernd-Ludwig Wenning, Carmelita Görg

Communication Networks,
Otto-Hahn-Allee - NW1,
University Bremen,
28359 Bremen,
Germany

{mab|wenn|cg}@comnets.uni-bremen.de

Abstract. Integrating simulation of communication networks with a logistical networks simulation is investigated taking advantage of object-oriented languages' features and design patterns. In this paper we present a simulator for logistic processes and the way how it is extended to enable combined simulation of logistical and communication networks. The aim of this integration is to measure the influence that an improved communication has on logistical transport processes.

Introduction

The dynamic and structural complexity of logistical networks makes it very difficult to provide all information necessary for a central planning and control instance. It requires, therefore, adaptive logistic processes including autonomous capabilities for the decentralised coordination of autonomous logistic objects in a heterarchical structure. The autonomy of logistic objects such as cargo, transit equipment and transportation systems needs to be supported by wireless communication networks. These technologies and others permit new control strategies and autonomous decentralised control systems for logistic processes. In this setting, aspects like flexibility, adaptivity and reactivity to dynamically changing external influences while maintaining the global goals are of central interest.

A simulator was developed in the Collaborative Research Centre 637 "Autonomous Cooperating Logistic Processes - A Paradigm Shift and its Limitations" (CRC 637) (see [1]) for simulating logistic networks and their components and to investigate the autonomy of the components. This simulator however had the communication implicitly integrated and it is hidden in the algorithms. As the communication is of high importance to the autonomous logistics it needs to be studied and therefore the implicit communication needs to be made explicit and measureable.

In order to enable the measuring and enforcing the explicit communication, features of object oriented programming languages, such as access control and templates, were chosen as the simulator is implemented in C++ using the Communication Networks Class Library (CNCL, available at [2]). CNCL provides the basis for discrete event simulations. In the following sections we will discuss the advantages and problems of these methods. Furthermore we integrated several design patterns (as suggested by [3]) into the simulator and will point out what design patterns can do for a structured modeling of systems.

Implementation

Access Control

The access to member functions of classes can be restricted. The access descriptors are `public`, `protected` and `private`. Methods with `public` access descriptor can be called from other classes, methods with

`protected` access descriptor can be called from this class and its derivatives, methods with `private` descriptor can be called only from this class.

In order to circumvent these restrictions for some classes they may be declared as `friend`. Additionally it is also possible not to declare whole classes as `friend`, but just make methods `friend`. For more information on access control, refer to [4].

Templates

Templates are an advanced concept of object-oriented programming languages for reuse of code. When the same methods are available for multiple objects, the method is a candidate for a template. In other words, when an algorithm is not only valid for one but multiple datatypes, it makes sense to use templates. Templates are also called type parameterization. The most well-known application of templates is the C++ standard library, formerly called standard template library (STL), see [5].

Problems with Templates

Currently there are some problems with the compilers' support for templates at least in the GNU compiler collection ([6]). The G++ versions up to 3.3 can't parse templates with a template return value. This problem is solved in version 3.4 of the G++.

But as of version 3.4 it is still not possible to put the implementation of a template function into the `.cpp` file, it has to be in the `.h` file. This leads to problems with mutual inclusion of header files. Consider the files `a.h`, `a.cpp`, `b.h` and `b.cpp`, where `a.cpp` includes `a.h` and `b.h` and `b.cpp` includes `a.h` and `b.h`. When templates are needed in both classes, as for our setup in the next section, and the template methods in both classes refer to the other class, we have mutual inclusion in the header file, which can't be resolved by current G++ compilers. This might be resolved by refactoring at least one class to two classes which breaks the mutual inclusion of the header files.

Application of Access Control and Templates

The logistical simulator mainly consisted of classes for logistical objects `Vertex` (e.g. stores, factory), `Edge` (e.g. roads, tracks), `Vehicle` (e.g. lorries, trains) and `Package` (e.g. goods). The communication between them was done by directly calling methods of the other classes, thus hiding the communication that takes place between the components of the logistical network.

The first step to explicit the communication was to integrate `CommunicationUnits` into the logistical components. The `CommunicationUnit` might be a UMTS card, WLAN adapter, GSM phone, Bluetooth Dongle or similar devices. As the logistical components could have more than one `CommunicationUnit`, we introduced a `CommunicationUnitManager`, which selects one of the `CommunicationUnits`. Furthermore to connect the `CommunicationUnits` and their Manager, we created a `MetaCommunicationUnitManager`, in which all the functionality to implement the communication is assembled.

The second step was to use the access controls of C++, as described in the earlier section. The access is restricted to methods that are providing information (implicit communication) to the `protected` scheme. This gave rise to compiler warnings, which show the places in the simulator where implicit communication is used.

Introducing template functions as `communicateWithVertexReq()` into the `CommunicationUnits`, `CommunicationUnitManager` and `MetaCommunicationUnitManager` was the third step. This function has the access rights and friends set in such a way that the logistical classes can only access the `CommunicationUnitManager`, this class again can only call the `CommunicationUnit`, which itself in turn calls the `MetaCommunicationUnitManager`. This gives us the possibility to access and restrict the Communication in all three instances. The `MetaCommunicationUnitManager` will finally go back to the `CommunicationUnitManagerRec` (receiving side) and `CommunicationUnit` of the appropriate Vertex by calling the template method `communicateWithVertexInd()`. The reason for the split-up of `CommunicationUnitManagerRec` and `CommunicationUnitManager` is that templates need to be defined in the header in combination with the mutual inclusion of `CommunicationUnitManager` and `MetaCommunicationUnitManager`. By the division of the functionality of the sending and the receiving side, we can break the mutual inclusion and work around the compiler's deficiencies, as pointed out in the previous section.

In order to call a method that includes communication, the method's name and its parameters are given as parameters to `communicateWithVertexReq()`. The next section explains the way the MetaCommunicationUnitManager will map the name of the method to the method itself.

Communication Function Database

The communication function database provides access to the functions that are implicitly containing communication for the classes that are allowed to communicate. This way we prohibit the usual way to call a function, in order to intercept the function call at a single point in the simulator for gathering statistics regarding the function calls.

The communication function database is a class called `CommFunctionDB` and the UML diagram is shown in Figure 2. The class consists of two access methods: `registerMethod()` to register a function in the database and `getMethod()` in order to return a function pointer to the function. The functions are stored in a map container with the function name as the key to retrieve the function.
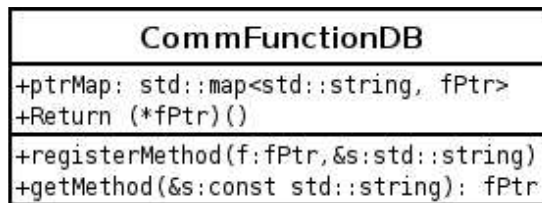
CommFunctionDB
+ptrMap: std::map<std::string, fPtr> +Return (*fPtr)()
+registerMethod(f:fPtr,&s:std::string) +getMethod(&s:const std::string): fPtr

Figure 1 UML diagram of the CommFunctionDB

Design Patterns

Design Patterns are elements of reusable object oriented software. They are recurring class structures in programmer's work of years and have been ensembled by Gamma et. al. in [3].

One such design pattern is the Singleton. The Singleton ensures that a class has only one instance and provides a global point of access to it. The UML diagram is shown in Figure 2. It is important for some classes to have only one instance. For example, in the combined simulation of logistics and communication, there should be only one transportation network. The way this is done, is to limit access to the classes constructor, but to add an access method `getInstance()`. This method returns the object, if the object exists, otherwise the method creates and returns the object. In simulation, this ensures that the class being a singleton cannot be created twice, thus removing a possible source of errors.
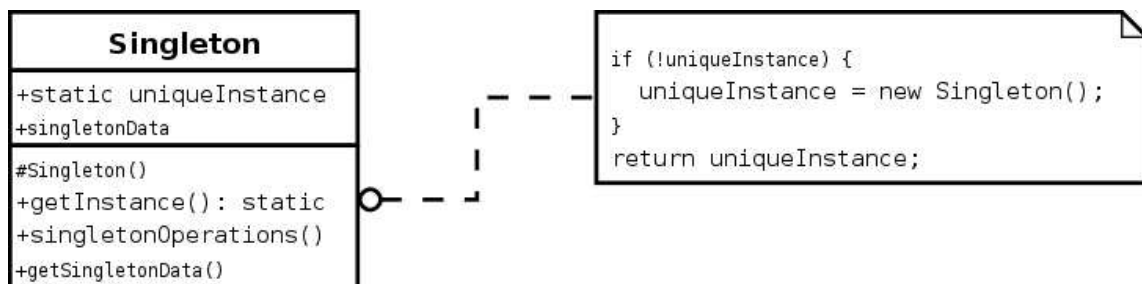
Singleton
+static uniqueInstance +singletonData
#Singleton() +getInstance(): static +singletonOperations() +getSingletonData()

```
if (!uniqueInstance) {
    uniqueInstance = new Singleton();
}
return uniqueInstance;
```

Figure 2 UML diagram of the design pattern Singleton

In the combined simulation presented in this report, the singleton pattern is used frequently combined with standard containers, such as `list<>` or `vector<>`. The resulting pattern is called `Manager`. Depending on the data type stored in the containers, e.g. `Vehicle` or `Packet`, the `Manager` is then called `VehicleManager` or `PacketManager`. The managers provide global access to instances of different types. This way the instances are always accessible at the same location.

Simulation

The simulation, that is based on the aforementioned object-oriented features and design patterns, is carried out using a scenario, that consists of the biggest german cities, which are interconnected by the major freeways in Germany, depicted in Figure 3. The distances between the cities are according to the real distances using the freeways. The need for transport is created in sources attached to the cities; the amount of this need is in accordance to the size of the city. The offer of transport means (i.e. Vehicles) is as well corresponding to the size

of the city. The packets, created in the sources, can have different destinations, but the packets of one source have the same properties, such as e.g. size or the type of communication systems. The destinations are chosen randomly, before the simulation is run. From each city packets are created for 8 different destinations. The way that the packets are discovering their ways through the network, is based on an algorithm derived from the Ad-hoc On-Demand Distance Vector (AODV) routing algorithm [7]. The loading strategy acts in such a way that it loads as many packets as possible for one destination onto the vehicle. The destination of the vehicle is determined by the destination of the majority of packets on the vehicle.
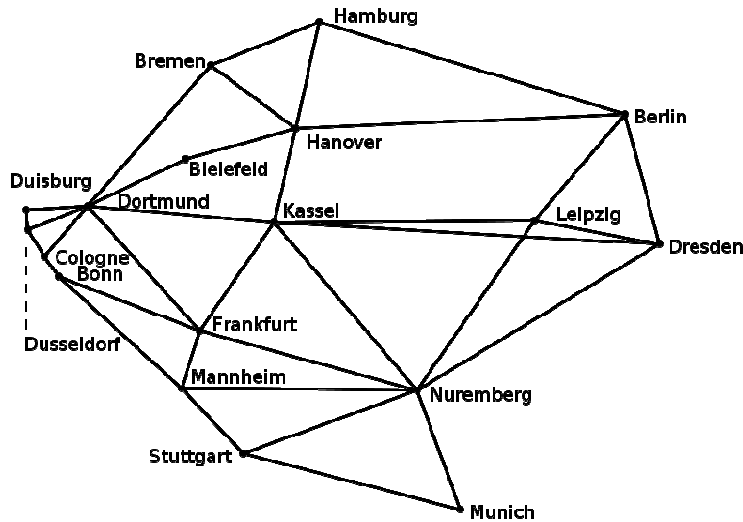
Figure 3 Simulation scenario (transport network)

Further parameters can be taken from Table 1.

Parameter	Value	Unit
Number of nodes	18	
Number of edges	69	
Sources per node	2	
Destinations per source	4	
Transport capacity /Transport means	60	Packets
Transport velocity	120	km / time unit
Transport creation rate / Source	17...1	Packets / time unit
Number of transport means	71	

Table 1 Simulation parameter

Resulting from the modifications to the earlier version of the simulator, it is now possible to intercept and record the communication frequency and volume. For the routing algorithm mentioned earlier the following functions are called using the function database:

- getting and setting of the destination of vehicles,
- getting and setting of the destination of packets and
- routing information exchange between the cities.

Table 2 shows some of the results based on the parameters given in Table 1. At a first glance the absolute communication frequency seems rather high, but the simulated time is 30000 time units long and the number of delivered packets is high as well. This results in an average netto communication volume per communication act of about 5 byte. This communication volume does not include addresses, signalling or coding. An average communication frequency per communication unit of about 42 byte results in an average communication volume per communication unit of close to 200 byte.

Parameter	Value	Unit
Absolute communication frequency	213,352,114	
Communication volume	962,518,244	byte
Average communication volume per communication act	4.5114	byte
Total number of transported packets	~5,000,000	
Average communication volume per communication unit	192.5036	byte
Average communication frequency per communication unit	42.6704	

Table 2 Simulation results

Conclusion

Object-oriented languages with template mechanisms such as C++ enable a way to a modular design of simulators. Using templates, design patterns and access control it was possible to enhance a simulator for logistical networks with functionality to simulate communication networks in such a way that they are of interest in logistical networks.

References

[1] Collaborative Research Centre 637 "Autonomous Cooperating Logistics Processes: A Paradigm Shift and its Limitations", http://www.sfb637.uni-bremen.de/index.php?id=8&L=2
[2] Communication Networks Class Library (CNCL), http://www.comnets.uni-bremen.de/typo3site/index.php?id=31
[3] E. Gamma, R. Helm, R. E. Johnson, J. Vlissides, Design Patterns – Elements of Reusable Object-Oriented Software, Addison Wesley. 1995.
[4] B. Stroustroup, The C++ Programming Language (Special Edition). Addison Wesley. February 2000.
[5] N. Josuttis, The C++ Standard Library: A Tutorial and Reference. Addison-Wesley. 1999.
[6] The GNU compiler collection. http://gcc.gnu.org/
[7] C. E. Perkins, Ad Hoc Networking, Addison-Wesley 2001.