Dartmouth College Computer Science Technical Report TR2001-405

DaSSFNet: An Extension to DaSSF for High-Performance Network Modeling

Mehmet Iyigun ‘01
Advisor: David M. Nicol

June 1, 2001
Dartmouth College

Abstract

Scalable Simulation Framework (SSF) is a discrete-event simulation framework providing a unified programming interface geared towards network simulation. Dartmouth SSF (DaSSF) is a C++ implementation of SSF, designed for simulating very large-scale multi-protocol communication networks. As of the latest release, DaSSF lacks many features present in SSF and this prevents it from achieving mainstream use. To alleviate this shortcoming we designed and implemented DaSSFNet which extends DaSSF to the levels of functionality found in SSF. In this paper, we show that DaSSFNet and SSFNet are identical in operation given the same input. We also show that DaSSFNet is about twice as fast and has one third the memory consumption of SSFNet when simulating identical networks. Therefore, we argue, that the DaSSF simulation package with DaSSFNet now offers a viable alternative to SSF in high-performance network simulation.

Introduction

Network modeling and simulation is an indispensable tool for evaluating network applications in today's connected world. Being able to install and run protocols in an arbitrary network topology allows developers to efficiently validate and analyze their protocols, thus reducing development time and cost.

1. SSF and SSFNet

Scalable Simulation Framework (SSF) is a general-purpose discrete-event simulation framework providing a unified programming interface geared towards network simulation. This programming interface is an object-oriented API for performing process-oriented discrete event simulation [1]. The SSF object model provides the following objects: *Process*, *Entity*, *Event*, *outChannel*, and *inChannel*. In this model, each *Entity* object owns a number of *Processes* that are executing in its context. *Entities* may also contain *inChannels* and *outChannel* that are used for communication. When an *Entity* A wants to communicate with *Entity* B, it creates and writes an *Event* into one of its *outChannels* that is *mapped* to B's *inChannel*. If B has a *Process* waiting on the *inChannel*, it will read and process the *Event*. Each mapping between an *outChannel* and an *inChannel* may have a certain delay. Moreover, each write into an *outChannel* may specify an additional delay. That way, *Entities* can cause the simulation time to advance when sending *Events* to each other. For more complete information on the SSF object model see [2].

Although the *inChannel* and the *outChannel* objects in the SSF object model might remind one of network interfaces or links, SSF really doesn't know anything about simulating networks. As far as the SSF API is concerned, there's no difference between using the same objects to simulate a molecule or a network. What really enables SSF to shine in the network simulation field is SSFNet which provides its own object model and framework on top of SSF specifically aimed at simulating networks. SSFNet provides objects representing network elements such as Machines, Links, Interfaces and Protocols. Each of these objects can be easily configured through Domain Modeling Language (DML). DML is a very simple, but powerful language consisting of attribute-value pairs. We'll see more about DML in Section 3.1.1. The DML interface allows the simulator to easily create arbitrary networks and feed them into SSFNet for simulation. This feature alone makes SSF and SSFNet a very attractive choice as a network simulation package.

2. DaSSF and DaSSFNet?

Since the core SSF API is public, it is possible to implement the same API using a language other than Java. Dartmouth SSF (DaSSF) is a C++ implementation of the SSF API. The DaSSF objects corresponding to the SSF ones are, *SSF_Process*, *SSF_Entity*, *SSF_Event*, *SSF_OutChannel*, and *SSF_InChannel*. In addition to the core SSF objects, DaSSF provides what it calls *SSF Extensions*. These extensions are aimed towards achieving higher performance or better usability while providing the same features as their SSF counterparts. For instance the *DirectInChannel* class in DaSSF is an extension

of the *SSF_InChannel* class, which doesn't require a process to be waiting on the channel in order to receive events. Instead, the *DirectInChannel* calls a function when an event arrives at the channel, thereby eliminating the need for a process wait. See [3] for a very detailed DaSSF manual.

Although DaSSF is at least as powerful as SSF as a discrete-event simulation kernel, it lacks crucial features of SSFNet such as DML-configurability. Because of this, DaSSF has not achieved the mainstream use that SSF has enjoyed for a long time. To alleviate this shortcoming, we designed and implemented DaSSFNet, which extends DaSSF to the levels of functionality found in SSF. Similar to SSFNet the DaSSFNet framework provides a complete suite of network elements such as Hosts, Routers, Links and Protocols all of which are configurable through DML.

3 Goals of DaSSFNet

The main goal behind DaSSFNet is to make DaSSF a viable alternative to SSF as a network simulation package. This objective can be broken down into several subgoals.

3.1 Configurability

Being able to easily create and configure networks is extremely important because it allows the user to be a lot more efficient. For this reason, support for DML configuration is a prime goal for DaSSFNet.

3.1.1 Domain Modeling Language (DML)

DML is a simple language that can be used to describe anything that has properties. Each DML expression is a list of whitespace separated key and value pairs. Although keys (property names) can only be strings, the value can either be a string (a simple property value) or it can be another DML expression (a property value that has more properties). Here's what the DML grammar looks like:

```
DML          ::=  (space* Attribute space*)+
Attribute    ::=  Key space+ Value
Key          ::=  Name | ReservedName
Name         ::=  ^[^_][a-zA-Z0-9_-]*
ReservedName ::=  ^_[a-zA-Z0-9_-]*
Value        ::=  String | \[ DML \]
String       ::=  [^(space)\]\[]+ | "[^"]+"
space        ::=  [ \t\n]
```

For instance the following could be a DML expression describing a car:

```
Car [
  Name DreamCar
  Wheels 4
  Seats 6
  Engine [
```

```
    Volume 3000
    HP 200
  ]
]
```

In order to use DML to describe networks, SSFNet has selected certain strings such as `net`, `host`, `router`, `interface` to have special meanings. For example, the following DML expression describes a host with 3 interfaces and a protocol stack composed of four protocols:

```
host [
  id 1
  interface [
    idrange [from 1 to 3]
    bitrate 10000000
    latency 0.001
  ]
  graph [
    ProtocolSession [name Netscape]
    ProtocolSession [name socketMaster]
    ProtocolSession [name TCP]
    ProtocolSession [name IP]
]
```

We will look at the exact range of DML attributes supported by DaSSFNet in Section 5.

3.2 Extendibility

DaSSFNet design should allow anyone to easily make modifications to the code in order to add features and improve usability. It should not make unwarranted assumptions about what its users expect or confine the user to a restricted set of features. For example, one feature a developer might want to add to DaSSFNet is a framework for automatically taking latency or throughput measurements at various points in the simulated network. The interface/link design of DaSSFNet should allow problem-free addition of such features.

3.3 Ease of Protocol Development

The reason most people use a network simulation package is to test their applications on various network configurations. Therefore it should be straightforward to develop protocols that run on DaSSFNet. Since these protocols might eventually be used on an actual operating system running on some network, the translation of the code between the DaSSFNet and OS frameworks should be relatively easy, so that the user needs to change the minimum amount of code to move from one to another. Furthermore, the protocol stack design should allow arbitrary composition of protocols with respect to one another. Conforming to the principles in Section 3.2, the protocol stack should not assume the existence of certain protocols in the stack to be functional.

3.4 High Performance, Low Overhead

Being able to simulate bigger networks over a longer simulation time is advantageous for protocol developers because it allows them to analyze the long-term behavior of the software in a more realistic setting. Therefore, the network application has a better chance of surviving when deployed in the Internet. So, a good network simulation package must strive to achieve the highest simulated time / real time ratio and at the same time keep a low memory profile. The high simulated time / real time ratio means that the user can simulate his network for a longer simulated time and the latter enables the user to simulate bigger network models.

4. Design of DaSSFNet

DaSSFNet basically includes a number of C++ classes that model the behavior of some network element. Some of these, like the SSF_Protocol class, are meant to be used by protocol implementers to derive from and others are meant to be used stand-alone.

4.1 SSF_Protocol and ProtoGraph

The SSF_Protocol and ProtoGraph classes together describe the protocols running on a simulated host. Every DaSSFNet protocol must be derived from SSF_Protocol and ProtoGraph is just a container for SSF_Protocols. Upon thinking about how different protocols interact with each other, we came to the conclusion that the protocols running on a host constitute a Directed Acyclic Graph (DAG) rather than a stack. This is because protocols like IP can have more than one protocol on top of it while every protocol always has only one protocol below. Since protocols need to send packets both to their parents and to their child, the SSF_Protocol class contains a list of parents and a child protocol pointer.

To enable the transfer of data between protocols, SSF_Protocol contains two pure virtual functions to send and receive data to/from a protocol:

```
// Send Data down the protocol stack.
// MUST BE IMPLEMENTED BY THE DERIVED CLASS.
virtual int send (msglist* data,
                  general_data* options=NULL,
                  uint32 oplength=0) = 0;

// Receive Data up the protocol stack to the application
// MUST BE IMPLEMENTED BY THE DERIVED CLASS.
virtual int receive (byte* data, uint32 length,
                     general_data* options=NULL,
                     uint32 oplength=0) = 0;
```

In order to give the protocols the maximum freedom in exchanging messages with different context information, each call to `send()` and `receive()` contains `options` which can be any data relevant to the particular communications. In the next section, we will see an example of the use of this feature.

One important requirement that DML-configurability brings about is the ability to instantiate protocols on a host give the protocol's name. This is no problem in the Java implementation because Java supports the creation of classes by name. In C++, however, this is more challenging. In order to be able to implement this crucial feature, we require each protocol to "register" with the protocol graph statically. In doing so, each protocol has to associate its name with a "creation function" which, when called, will create an instance of the protocol. The details of this procedure are hidden behind the `REGISTER_PROTOCOL` macro and the protocol implementer only needs to implement a static "creation function" and pass the function pointer to the `REGISTER_PROTOCOL` macro along with the protocol name in global scope. Internally, the ProtoGraph class contains a static hashtable that maps protocol names to creation functions. When the `REGISTER_PROTOCOL` macro is called by a protocol at global scope, the <protocol

name, creation function> pair will be added to the hashtable and consequently all protocols compiled into the DaSSFNet executable will have registered before main() starts executing. Therefore, when the ProtoGraph needs to create a protocol given its name, it can just look it up in the hashtable and call the creation function. Before creating any protocol, the ProtoGraph class instantiates the Hardware class, which is an internal protocol, which is always at the root of the protocol DAG. The role of the Hardware class will be discussed in the next section.

In order to support DML configuration and enable protocols to do initialization at different phases of the simulation, SSF_Protocol class contains two pure virtual functions:

```
//
// After a protocol is created, it will be given a DML_AttribList which
// corresponds to the DML attributes specified in the DML file for this
// protocol. The protocol should read, verify and apply the attributes.
//
virtual int Configure(DML_AttribList* config) = 0;

//
// This function will be called after the entire network is read in and
// Configure()d and after all the links have been connected,
// but before the simulation starts running. The protocols
// can do any kind of initialization they need which requires the network
// to be connected.
//
virtual int Initialize() = 0;
```

As explained in the function comments, the `Configure()` function will be called with the DML attribute list belonging to this protocol so that the protocol can configure itself. In order to give the protocol one last chance to initialize before the start of the simulation, we also provide the `Initialize()` function that gets called after the network has been connected and IP addresses have been assigned. For instance, the static OSPF protocol that we implemented for DaSSFNet uses this callback to calculate its shortest paths.

4.2 Interface and Link

The Interface and Link classes together handle all the communication that happens during the simulation. They work together to ensure that network packets get delivered to the correct interface on a link and that each packet incurs the required delay given the bit rate and the latency on the interface and the link. In addition, the Interface class implements the interface internal buffer, which is used to store packets before they are sent out at the interface's bit rate. The implementation makes sure that the interface never sends data faster than its bit rate and that if the interface receives so much data that its buffer overflows, the extra packets are dropped. It is worth noting here that the interface buffer and bit rate checks are implemented only when sending data; so an interface can receive data at any rate. The assumption here (also made by SSFNet) is that an interface will always be connected to another of the same speed, so that implementing the buffer and bit rate at one end is sufficient.

DaSSFNet is not designed for the simulation of link layer protocols like Ethernet or Token Ring. As such, it needs to be able to do link layer routing when there are more than two hosts on a link (called a Local Area Network - LAN link). We implement this requirement at the Hardware class. Being a regular SSF_Protocol in the protocol graph root, the Hardware will receive any data sent down from the lowest layer protocol. Here, we require the lowest-layer protocol to place the routing table entry for the packet destination in the `options` parameter of the `send()` function. The Hardware class, takes the next hop field from the routing table entry, prefixes it to the message and sends it down to the appropriate link as specified in the routing table entry. When the packet finally gets to the Link class, the link looks at the next hop and resolves it to the correct interface by using a hash table, which it precomputes during initialization.

4.2.1 NHI Addressing

When a link is specified in DML, the following format is used:

```
Link [attach 1:2(3) 4:5(6)]
```

The 1:2(3) and 4:5(6) are the Net:Host:Interface (NHI) addresses of the interfaces that are on this link. The last two numbers in an NHI interface address are the machine and the interface id respectively. Any numbers before the last two are Net ids. NHI addresses of interfaces in the DML expression for the link are given relative to the enclosing Net. So 1:2(3) refers to interface id 3 on machine id 2, which is in Net id 1. Please refer to [4] for a full discussion on NHI addressing.

4.3 Machine

The machine class models both a host and a router. In fact, there's absolutely no difference between a host and a router from an implementation point of view. Frequently, however, hosts have only one interface while routers have many. The machine is basically a container for the protocol graph running on the machine and the interfaces. Being derived from an SSF_Entity, a machine is also a container for SSF_Processes. This is extremely useful from a protocol implementation point of view because protocols that need to generate traffic actively (like a web browser application) need to be SSF_Processes and they can use the machine (which is passed to their constructor) as an owner. Our previous design for protocols that need to generate traffic was to derive them from SSF_Entity, which also works, but consumes a lot more memory.

In addition to acting as a container for protocols and interfaces, the machine also provides utility functions for the protocols to query the interfaces and links the machine is connected to.

4.4 Net

The Net is not a physical network element, but it acts as a container for machines, links and other nets. The reason one uses a Net attribute in DML[1], is to group machines

[1] One outermost Net is mandatory in DML. We're talking about inner Nets here.

or other nets sharing some common features. In the case of DaSSFNet, the common feature that the elements of a Net share is the IP address prefix; Nets in DaSSFNet are implemented as subnets and during IP address assignment, each Net gets its own IP prefix.

Since every element of the network is enclosed (at some level) in a Net, we also use Nets to resolve NHI addresses to Nets, Machines or Interfaces.

5. DML Support in DaSSFNet

This section is a complete description of the DML attributes supported by DaSSFNet. It only includes those attributes that are understood by DaSSFNet and not the ones specific to protocols.

Attribute	Enclosing Attribute(s)	Function
id <int>	Net, Host, Router, Interface	Assigns an integer id to the enclosing attribute
idrange [from <int1> to <int2>]	Net, Host, Router, Interface	Duplicates the enclosing attribute with ids ranging from <int1> to <int2>
Net	Net or nothing	Creates a Net
Host	Net	Creates a Host
Router	Net	Creates a Router
Interface	Host or Router	Creates an Interface
Link [attach <NHI1> … <NHIn>]	Net	Creates a Link that connects the Interfaces that the NHIs resolve to
Graph	Host or Router	Creates a Protocol Graph
ProtocolSession [name <str>]	Graph	Creates a protocol with the given name.
bitrate <int>	Interface	Defines the bit rate of the interface in bits/sec
latency <float>	Interface	Defines the internal latency of the interface in seconds.
buffer <int>	Interface	Defines the internal buffer size of the interface in bytes.
flaky <float>	Interface	Causes the interface to drop packets with the given probability.
no_queue <true|false>	Interface	Causes the interface not to use its queue so that packets don't incur any queuing delay.
delay <float>	Link	Defines the delay of the link in seconds.
nhi_route [dest default]	Host or Router	Defines a default route for this machine
interface <int>	nhi_route	Specifies the target interface for this default route by giving its id.

next_hop <NHI>	nhi_route	Specifies the next hop interface for this default route. Only required if the interface is on a LAN link.
Traffic	Outermost Net	Specifies the traffic pattern that traffic generator protocols should use.
Pattern	Traffic	Specifies one traffic pattern
Client <NHI>	Pattern	Specifies the source clients for the traffic pattern. The NHI will be first resolved as a Net address. If it works all clients in that Net will be selected. Otherwise, it will be resolved as a host address.
Servers	Pattern	Specifies the servers that the clients in this pattern should connect to
nhi <NHI>	Servers	Specifies the NHI address of the server in a servers attribute.
nhi_range [from <NHI1> to <NHI2>]	Servers	Specifies a range of NHI addresses of servers in a servers attribute. The host id in NHI1 will be incremented until it reaches NHI2's host field and all the matching servers will be selected.
Port <int>	Servers	Specifies the port on the server that the clients should connect to.
Frequency <int>	Outermost Net	Not used in DaSSFNet because DaSSF has a floating-point event queue.
AS_status boundary	Net	Specifies that the enclosing Net is an Autonomous System.
OSPF_area 0	Net	Not used
Seed <int>	Outermost Net	Used as a seed for the drand48() random number generator.

6. Comparison of DaSSFNet vith SSFNet

6.1 Correctness

We have implemented DaSSFNet so that given the same DML description of a network, it will behave the same as SSFNet if the protocols are identical. It is virtually impossible to prove this assertion because there are so many different cases to test. Instead, we have performed some experiments that strongly suggest that DaSSFNet and SSFNet are functionally identical.

Method: We implemented a pair of very simple server/client protocols called SimpleClient and SimpleServer for both DaSSFNet and SSFNet. The client side of the protocol sends a greeting message to the server at a DML-configurable interval and includes the current simulation time in the packet. When the server responds with another greeting, it doesn't change the time in the packet. When the client receives the response from the server, it calculates the roundtrip time for the packet and adds it to a global variable and increments the number of packets received. At the end of the simulation, the average round-trip time is printed along with the average throughput (which is just the multiplicative inverse of the average roundtrip time).

We ran the SimpleClient/SimpleServer pair on several networks with different topologies for 100 simulated seconds:

- campus2.dml (see Appendix 1)
- usa1.dml (a.k.a BIGNETWORK. See Appendix 2)
- usa2.dml (2 usa1.dmls linked together)

Here are the results:

DML File	Average Packet Latency		Number of Packets Sent	
	SSFNet	DaSSFNet	SSFNet	DaSSFNet
campus2.dml	0.08237	0.082	456	456
usa1.dml	0.07631	0.076	24700	24700
usa2.dml	0.08427	0.084	57200	57200

Therefore, we see that the behavior of DaSSFNet and SSFNet (at least for these network configurations) is identical.

6.2 Performance

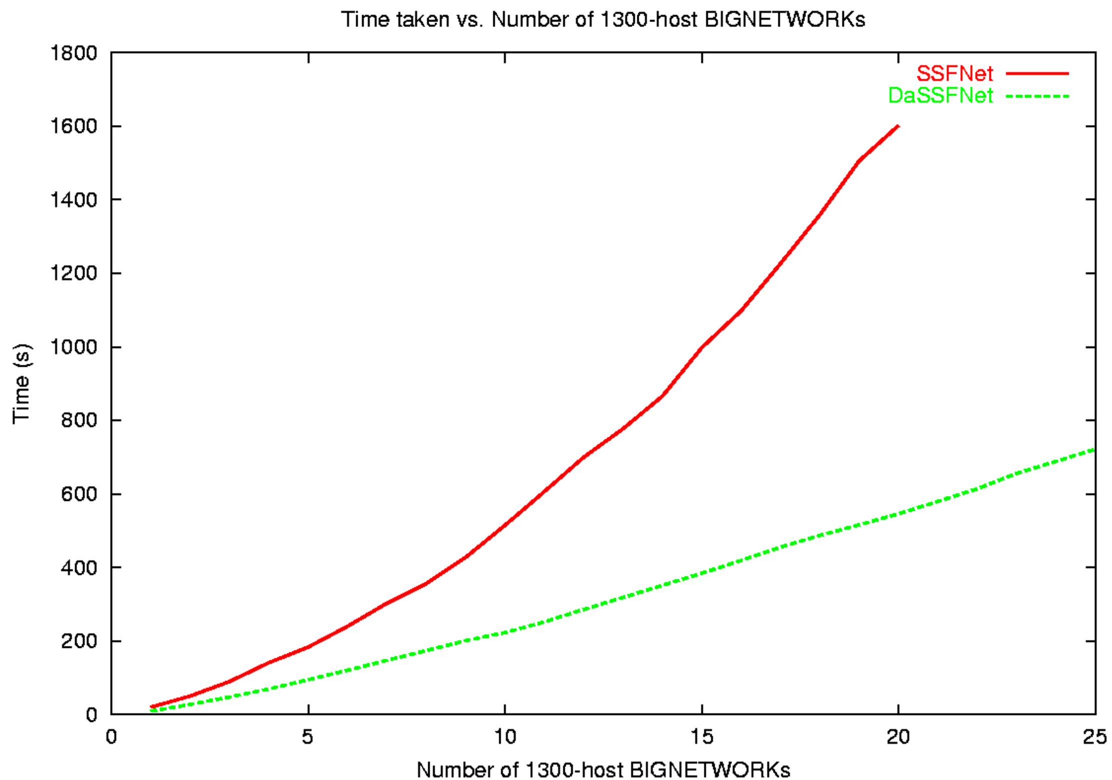
All our experiments were run on cairngorm.cs.dartmouth.edu which has 1GB of RAM and 4 500Mhz Intel PIII processors running on Linux 2.2.X.

6.2.1 Running Time

In order to measure the running time of DaSSFNet with respect to SSFNet, we performed two experiments:

- We simulated DaSSFNet and SSFNet on networks of increasing size. Since we didn't want protocol implementations to skew results, we used the above-mentioned SimpleClient/Server protocol pair which have virtually identical implementations in C++ and Java. We ran SSFNet and DaSSFNet for 100 simulated seconds on USAn.dml where n goes from 1 to 25. USAn.dml is composed of n usa1.dml's linked together in various ways. USA25.dml can be seen in Appendix 3.
- We ran DaSSFNet and SSFNet on USA10.dml for the following lengths of simulation time: 10, 50, 100, 400, 700, 1000 (s)

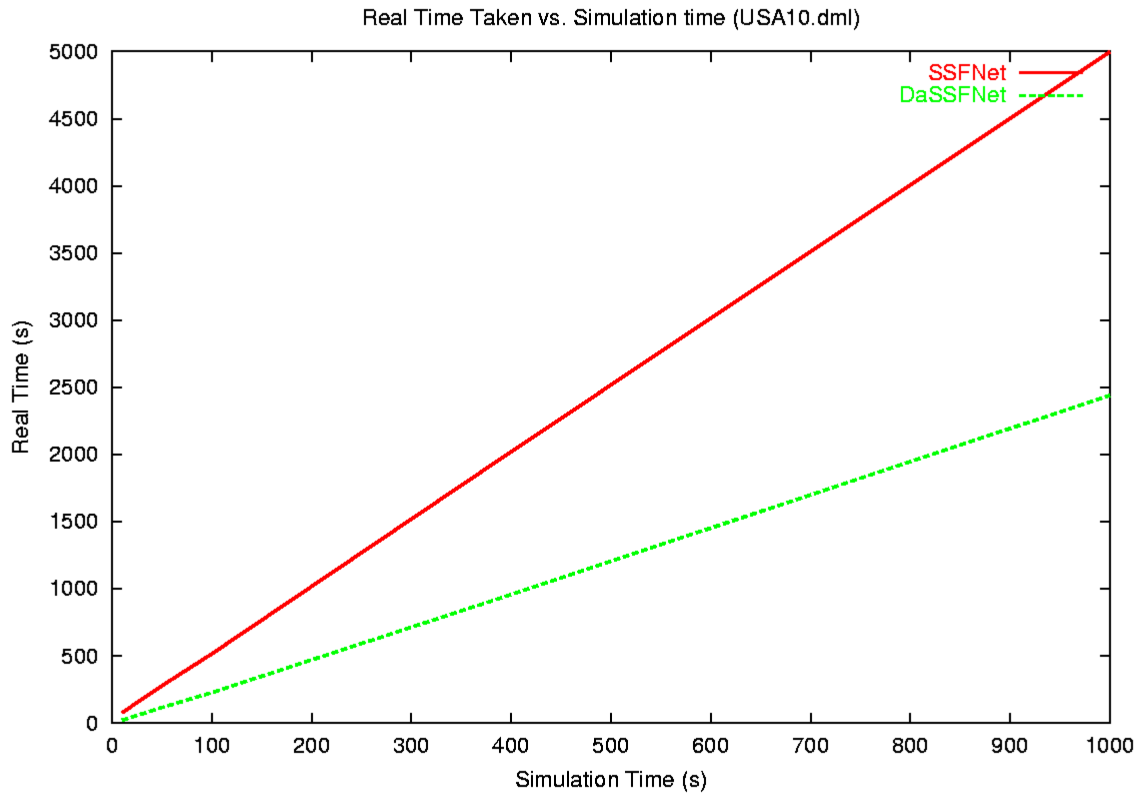
The following graph shows the relationship between running time and the size of the network. We could not simulate SSFNet on bigger networks than USA20.dml because the memory on the simulation machine was not sufficient.



This graph clearly shows that DaSSFNet has a minimum of 2X speed advantage over SSFNet and the ratio grows as the networks get bigger. Also notice that the running time of DaSSFNet is completely linear in the size of the network (as expected) whereas SSFNet diverges a little bit[2].

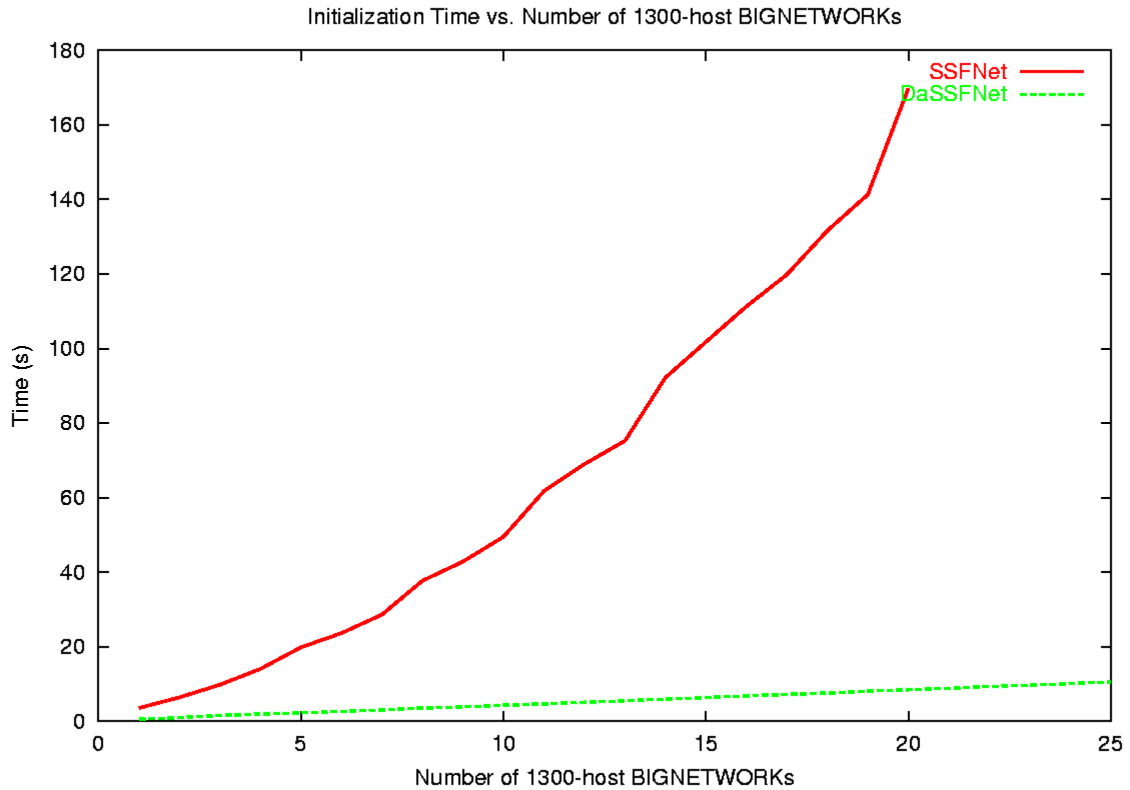
[2] This might be explained by the fact that towards the end of the simulation, SSFNet uses memory close to the limit set by –Xmx flag and therefore the garbage collector might be running more forcefully and thus taking more time.

The following graph shows the real time taken by each simulator versus the length of simulation.



The slope of the green line in the above graph is 2.44 and that of the red line is 4.97. This clearly shows that simulation time / real time ratio of DaSSFNet in this case is very close to twice that of SSFNet.

The following graph shows the relationship between the initialization time of the simulation and the size of the network being simulated. Initialization time is defined as the time it takes to actually start the simulation from the time the simulator is run.

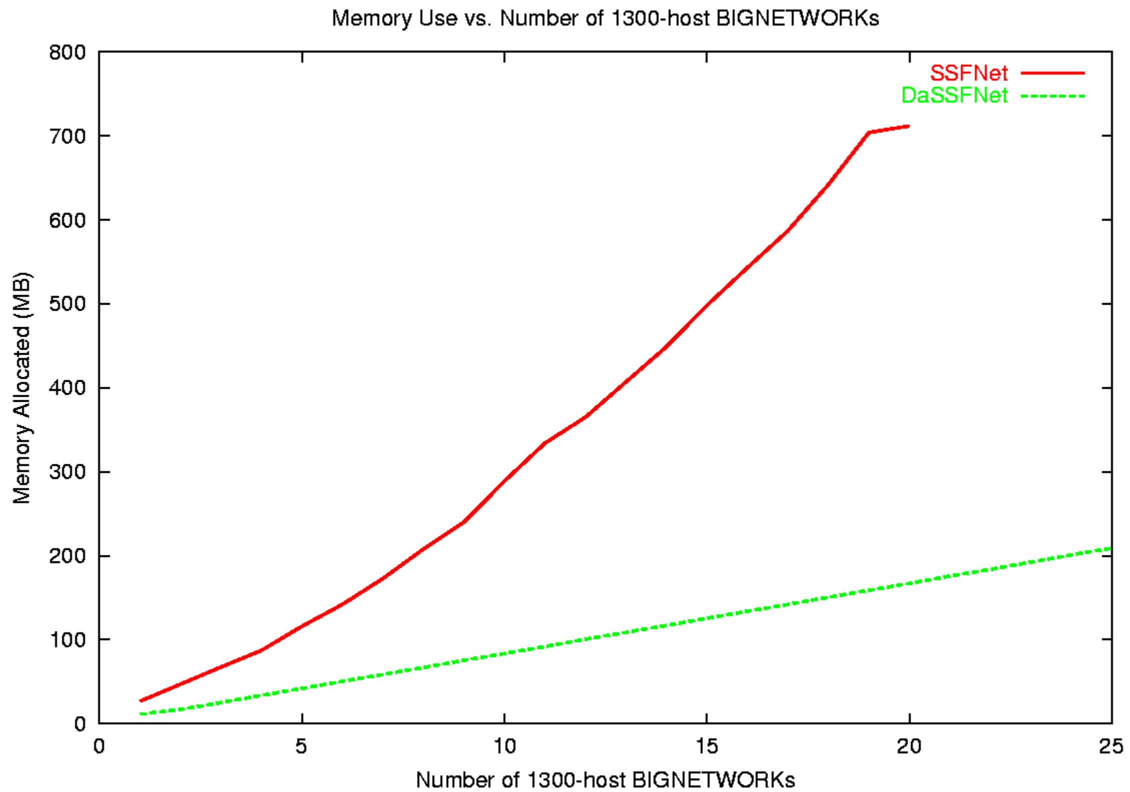


DaSSFNet has a big lead in the initialization time. Again, it's worth noting that DaSSFNet scales linearly with the size of the network (as expected) whereas SSFNet seems to be more like exponential.

6.2.2 Memory Use

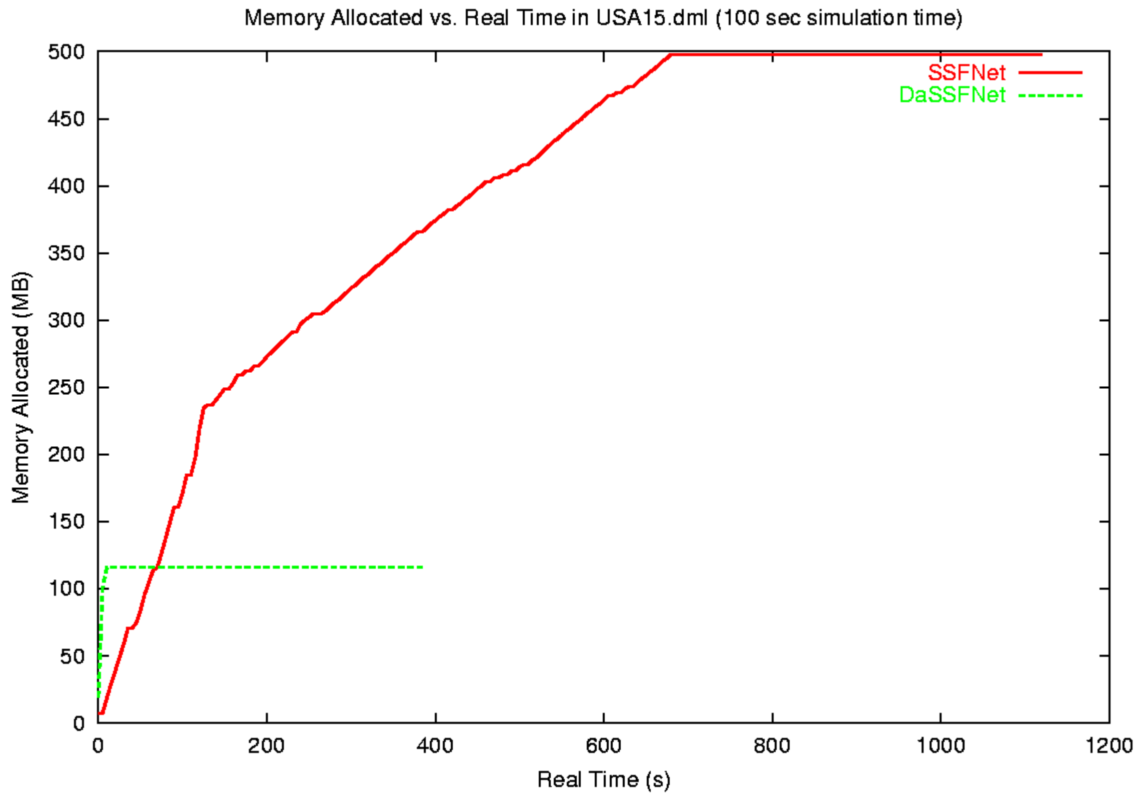
We ran the same experiments as in Section 6.2.1 and also recorded the memory usage of each simulator.

The following graph shows the relationship between amount of memory allocated and the size of the network.



The graph reveals that the memory consumption of DaSSFNet is about 1/3 of SSFNet. Moreover, the memory usage of DaSSFNet is linear in the network size as expected whereas SSFNet has some fluctuations here and there.

The next graph shows the memory allocation pattern of DaSSFNet and SSFNet while simulating USA15.dml.



This graph clearly shows that DaSSFNet's memory usage does not increase with time while SSFNet seems to allocate memory until more than halfway into the simulation. Also note that, in this case SSFNet uses about 4 times the memory of DaSSFNet and that DaSSFNet takes less than half the time to complete the simulation.

7. Conclusion

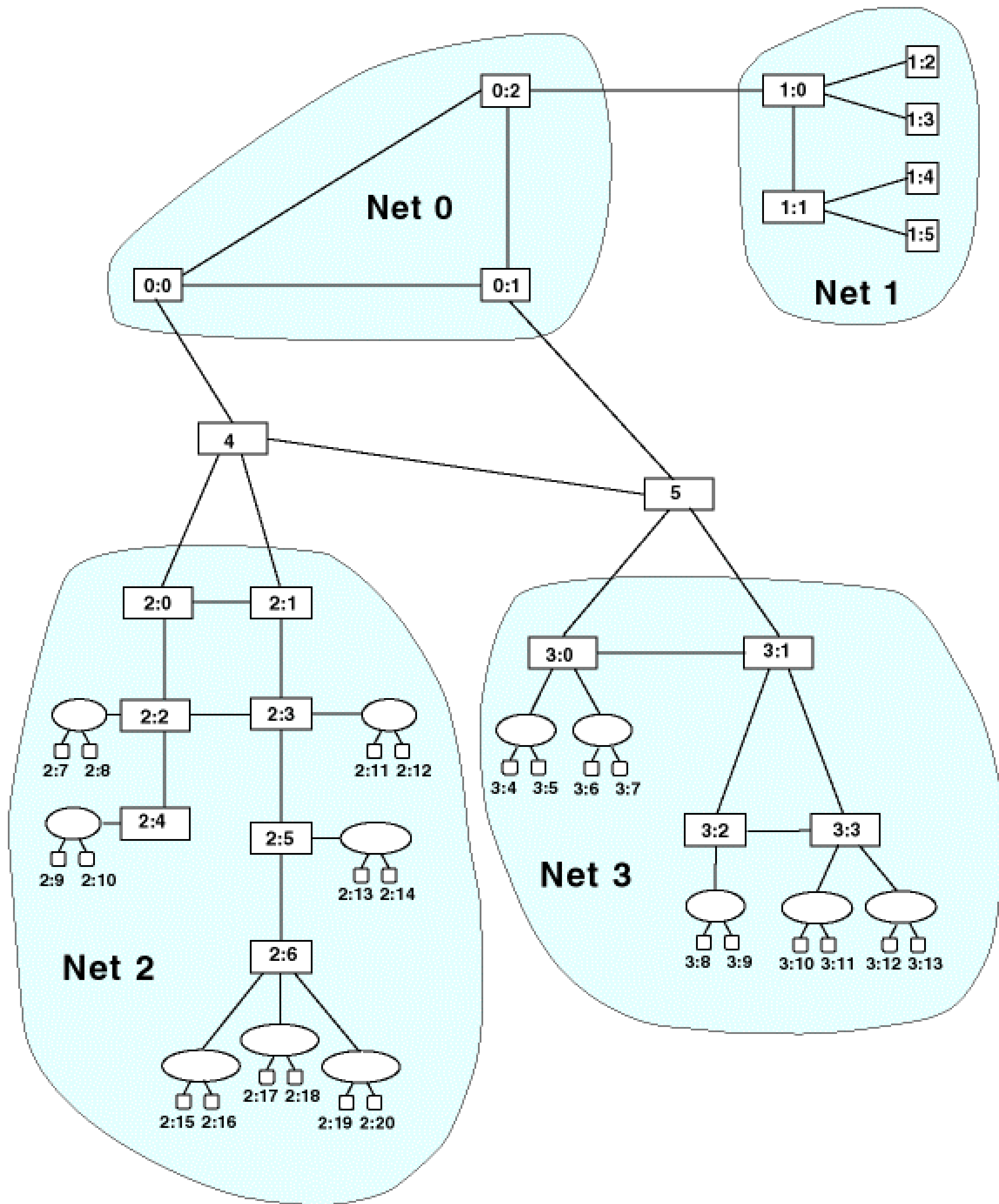
We have shown that DaSSFNet and SSFNet have identical behaviors when run on the same DML file if they have identical protocols. Moreover, we also demonstrated that DaSSFNet is superior to SSFNet in both running time and memory usage under identical conditions. With the implementation of some core internet protocols like TCP/IP, OSPF and BGP for DaSSFNet, it is now able to simulate most SSFNet network models with minimal modifications. Therefore, although still lacking in some features such as WWW clients/servers and DML-configurable measurement framework, the DaSSF simulation package with DaSSFNet now offers a viable alternative to SSF in high-performance network simulation.

BIBLIOGRAPHY

[1] http://www.ssfnet.org/SSFdocs/javaBinding.html
[2] http://www.ssfnet.org/SSFdocs/ssfapiManual.pdf
[3] http://www.cs.dartmouth.edu/~jasonliu/projects/ssf/papers/dassf-manual-3.1.ps
[4] http://www.ssfnet.org/InternetDocs/ssfnetDMLReference.html#nhi

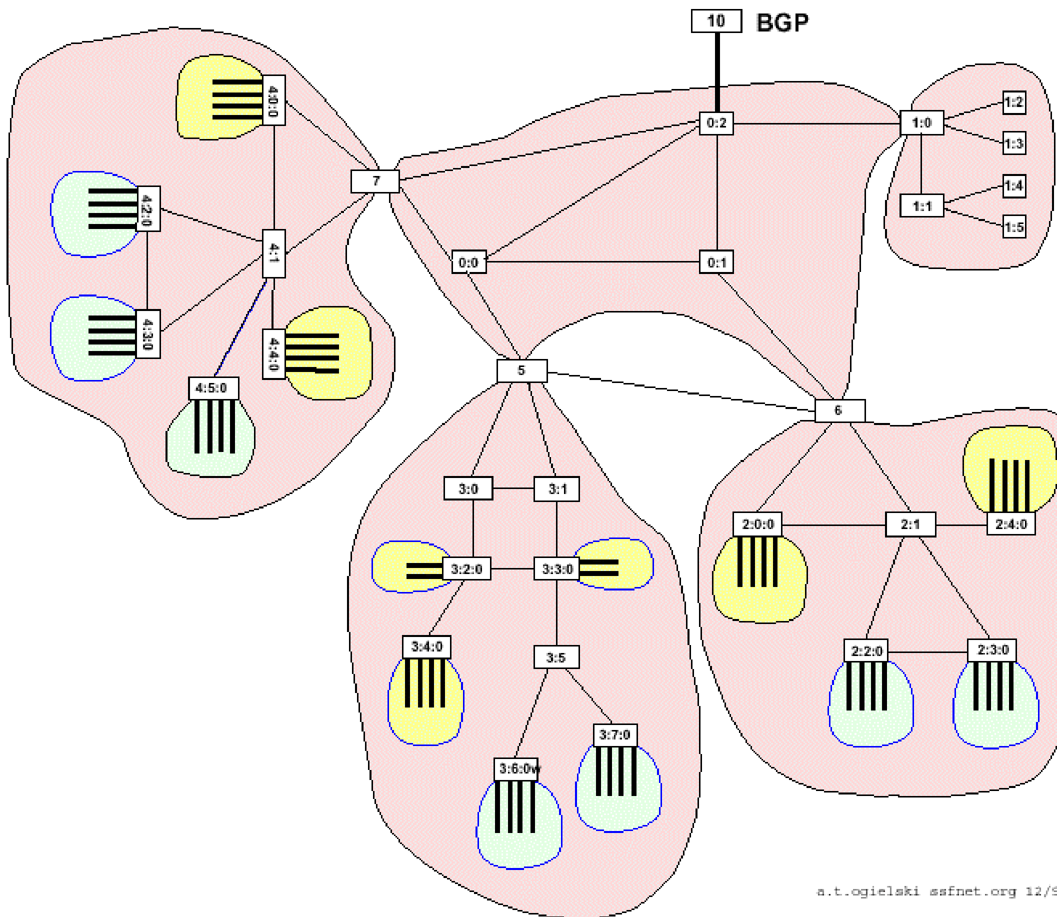
APPENDIX 1

campus2.dml



APPENDIX 2

usa1.dml (a.k.a BIGNETWORK)



APPENDIX 3

USA25.dml