University of Glasgow | School of Computing Science

ScaNS: Using Actors for Massive Simulation of Distributed Routing Algorithms

Helier Alexander Waite (0804196)

May 5, 2013

ABSTRACT

Existing simulators designed for the evaluation of distributed graph algorithms tend to be based on a threaded model of concurrency. This leads to scaling issues when simulating particularly large graphs such as Internet-like graphs or poor utilisation of resources when using large multiprocessor machines. Here, I present a simulator that uses actor-model concurrency to directly model the interactions between nodes in a graph during the simulation of a distributed algorithm. While more closely modelling the behaviour of an abstract network, the actor model brings performance advantages.

1. INTRODUCTION

The Internet is a large distributed system that is continuing to grow at a fast rate. Being able to forward information and share routing information over such a distributed system between any two arbitrary end-points is a essential to the operation of the network. The networks that comprise the Internet are called Autonomous Systems, or ASes. The ASes are the building blocks that are interconnected to create the internet. ASes are defined as being a single administrative domain, meaning they are administrated separately from most other ASes. These are usually run by Instenet Service Providers(ISP), and larger businesses.

The job of transferring reachability data between the subsystems in the Internet belongs to the Interdomain routing protocols. Interdomain routing protocols treats the internet as a series of nodes and links represented by ASes and the routes through the network that connects them. This node-link abstraction is known as the AS graph and is very important in interdomain Internet research.

The currently used interdomain routing protocol for the Internet is the Border Gateway Protocol(BGP)[14], which is used to maintain tables containing route data to other ASes at routers. BGP has some known scaling issues. Leading up to the year 2001, BGP routing tables exhibited exponential growth[11]. This represents a substantial problem as older backbone routers start to run out of memory to store new routes and lookups for existing routes start to take significant time. In the time between 2001 and 2012, the size of a full BGP graph has grown from just over 100,000 entries to 400,000 entries. In [7], the authors discuss some of the causes of this massive growth and conclude that the largest factor is that of address fragmentation, where every prefix requires an entry in the table but could be aggregated to a single entry. It was also noted in [9] that IPv4 exhaustion may have worked as a limiting factor in the growth of routing tables and that the wide-spread adoption of IPv6 may exacerbate the situation.

The growth of the network continues to be a problem due to the increased adoption of IPv6. IPv6 uses a much larger address space to mitigate the exhaustion issue found in IPv4. As the number of hosts using IPv6 increase, the size of routing tables will again be able to increase super-linearly as new IPv6 addresses are allocated.

Due to this, replacements for BGP need to be considered as the size of the Internet is guaranteed to grow. A full hardware upgrade of the backbone network would be very costly as well as largely infeasable due to the autonomous nature of the Internet. The development of such such protocols often require simulation of the new protocol over large Internet-like graphs to try and gain an idea of how the protocol behaves at these scales. Such a simulator has the requirements of being scalable in respect to graph size, as well as being able to make effective use of multicore and multiprocessor technology to achieve speedup.

This paper describes ScaNS, a scalable network simulator. The simulator was designed specifically for use with Internet-scale graphs and continues to perform well to multi-million object simulations. The simulator allows users to create a simulation of many million end-hosts running distributed algorithms such as new routing protocols, and study the behaviour in a high level way that can be used to reason about the behaviour of the algorithm. One of the main design features is that ScaNS uses actors as it's main unit of concurrency, rather than threads. This allowed us to test the hypothesis that as well as making the code easier to write, actors work well for this domain of problems as it maps directly to the underlying graph.

This paper makes several contributions. First, ScaNS is the first high-level routing simulator that I could find in literature that makes use of actor concurrency. I present the case for the use of actor model parallelism in relation to the class of simulators which deal with the simulation of communicating entities, such as in a distributed system. I present an evaluation of ScaNS against various metrics such as it's scalability and it's ability to directly model networks of objects. I also present a real-world use-case for the system, showing it's worth as a usable system.

The rest of this paper is structured as follows. Section 2 further discusses the problems surround the use of currently existing simulators for use with new interdomain routing algorithms. Section 3 discusses the design of ScaNS as well as some key concepts that it uses. Section 4 introduces low-level details of how ScaNS is implemented including details of how simulation objects work together using actors. Section 5 provides evaluations of the system to test it's performance characteristics using various metrics as well as test-

ing it's fit for purpose. Section 7 provides some more background information on simulators, the AS graph and routing algorithms relevant to this paper, and section 8 concludes and provides ideas for future work.

2. PROBLEM

There are two main classes of networking simulator that exist. protocol-level and abstract simulations of the network. The protocol-level class of simulators are focused on the simulation of the network various degrees of realism. These often have a focus on a particular aspect such as accurately modelling wireless communication channels. These are the sorts of simulators used for testing such aspects as traffic flow in smaller networks. The abstract class of network simulator concerns itself with abstracting away many of the details used in the protocol-level simulations in favour of a more abstract, instead of realistic, simulation. This type of simulator are used less for fine grained simulation as they lack the details required but are instead used for high level overviews of how larger systems work.

The study of distributed algorithms, such as routing protocols, is often less concerned with the physical implementations of protocol and are more focused on showing that the underlying algorithm works well. For this class of applications, standard network simulators do not work well. For these sort of simulations, the type of the channel, the installed network stack and other low-level network constructs become superfluous. Essentially what is required is a massively scalable distributed algorithm simulator. High-level simulation abstract away ideas like these and instead aim for a simpler "link and node" model. This allows users to experiment with new distributed algorithms without the hassle of having to define the protocol first.

For the simulation of low-level networks, several simulators already exist. These allow very fine grained control over many low-level aspects of a simulation node, such as being able to install a specific networking stack in a particular node. While this allows for very accurate simulation of small to medium topologies, the overhead of such network stacks and other micro-managed parameters for large graphs will become a restricting factor. These simulators are useful for fine grained traffic analysis of networks. This is not the sort of simulator that we need. The ability to specify these parameters will only complicate the code of the simulation and the granularity of the output from the simulation may be too fine.

While there are numerous low-level network simulators with various different features, there are very few high-level simulators. This is likely due to the need for high-level simulators being substantially lower than other types of network simulator as they have somewhat limited use-cases. This paper describes a high-level network simulator designed for the use with very high-level routing problems, allowing the user to disregard details such as defining network stacks and only focus on nodes, links and the algorithm.

An example of an experimental routing problem in this class is that of K-core decomposition [8]. The K-core decomposition of a graph can be used to show the connectivity properties of a graph. Prior work [16] has shown the the AS graph is an area where k-core decomposition would be relevant in the area of choosing landmark sets within the Thorup-Zwick compact routing scheme[17]. The algorithm is a centralised algorithm with global knowledge of the graph. A distributed version of the K-core decomposition algorithm is introduced in [12]. This distributed version of the algorithm could potentially be incorporated as part of a new routing scheme. Since evaluation of this new algorithm is currently a graph problem and not yet a protocol problem, a simulator for the evaluation of the distributed K-core decomposition algorithm would not require all of the detail of a low level network simulation and would instead wish to trade that for higher performance. Requirements for the simulation of this algorithm only require that the simulation maintain a set of nodes, a set of edges and signal when it is time for nodes to perform the algorithm.

3. SYSTEM DESIGN AND CONCEPTS

Here, I discuss ScaNS, the Scalable network simulator. The simulator was designed from the ground up to support the high-level study of the simulation of million-scale, distributed graphing problems with a particular emphasis on routing algorithms. In this section I discuss various concepts and design ideas important to the success of the system.

3.1 Actor Model Concurrency

ScaNS was designed to use the actor model of concurrency. The actor model, first introduced in [10], states that every unit of concurrency is called an actor. The only way to interact with an actor is to send messages to it. An actor is responsible for creating local decisions on it's behaviour based on a message that is has just received. These decisions could be performing computation, sending messages to other actors or creating new actors. This paradigm fits several real-world such as actors represent people and messages representing interaction, or in networks where actors represent entities in the network and messages represent data flowing between entities.

The actor model is in contrast to the threaded model of concurrency. The threaded model allows blocks of executable code to be run in different thread to attain speedup. The main communication method between threads is through shared memory regions protected by lock. The threaded model is know for being particularly hard to program correctly as well as it being very easy to introduce subtle bugs like deadlocks and inconsistent shared memory. The actor model deals with all of these by using the message passing mechanic. By not allowing functions to be called on actor objects, there is no way for an actor to become in an inconsistent state as there is when objects are modified from different threads. Also, since the actor model uses explicitly asynchronous message passing, this also removes the issue of locking as the entire system is now event-based instead of lock-based.

3.2 Scala

Scala[13] is a multi-paradigm programming language designed at EPFL in Switzerland by Martin Odersky, supporting simultaneous use of imperative, object-oriented and functional styles. It's main aim was to be a "component" language used to create component systems. It was designed to be fully interoperable with existing Java code while adding several functional programming additions and amending a few issues that the community saw in Java. Code written in scala is compiled down to Java virtual machine (JVM) bytecodes which is the mechanism for making it interoperable with existing Java code but also means that it can run

on standard JVM. Scala takes the object-oriented approach of Java a step further with it's idea of "everything is an object", or more formally a "unified type-system". "Traits" are scala's version of Java's "interfaces" but also allow behaviour to be defined in them. Traits were designed to be âĂIJmixed inâĂİ with class definitions to easily extend the functionality for a class such as mixing in a "debugging" trait in to any class to add a specific logging function. Traits were designed to be chained together such as having the class using debugging traits plus a trait that defines other component behaviour. The traits mechanism aids the "component" aspect of Scala as it allows users to define traits of often used functionality that can be easily mixed in to new classes when

Scala was chosen for this simulator for various reasons. It's heavy emphasis on being a component language allowed us to build a modular architecture which can be easily extended through the use of traits. The language has a substantial standard library to use as well as being fully compatibly with all of Java's standard library. This meant I already had all of the tools at my disposal to create such a system easily without having to re-implement standard constructs. Scala's interoperability with existing java libraries meant that users would also be able to use any existing java software in conjunction with the simulator to extend it's functionality. Scala has also always had an emphasis on providing an implementation of actor concurrency as part of it's standard distribution. This meant I could rely on having this robust base to build out entire actor system upon.

Scala's actor system is based heavily on the actor system presented in Erlang [5]. Erlang is a language designed for the use in robust telecommunication applications such as switching equipment at telephone exchanges. Erlang is a functional language that natively supports concurrency through message passing. Despite this, there are a few reasons that Erlang was not used for this project. Erlang being a functional language may not have been a good fit for the architecture I was trying to build due to immutable state. Since all of the simulation objects in the system need to maintain their own state, the paradigm of immutable state would become costly as scale where new objects would need to be created for every change. Erlang started as a modified version of Prolog and as such, has inherited many of it's syntax ideas. The syntax used by Erlang is one that is unfamiliar to myself and will be unfamiliar to many other developers who wish to use this application. This is in contract to the well known C-style syntax used by Scala.

3.3 Architecture

Figure 1 shows the overall architecture of the simulator. The system can be roughly split in to four sections: the host application, the simulator, the clock distribution and the graph.

3.3.1 The Graph

The graph section is the representation of the input to the simulator. The graph is modelled as everything being an actor, just in the same way as the physical world. Nodes, which implement the algorithms to be run in the simulation are represented by actors. This allows modelling of the receive, act, respond loop that is used in real hardware. When a node receives a message, it decides how to process it and then has the option to send other messages.

Links are also represented as actors within the system.

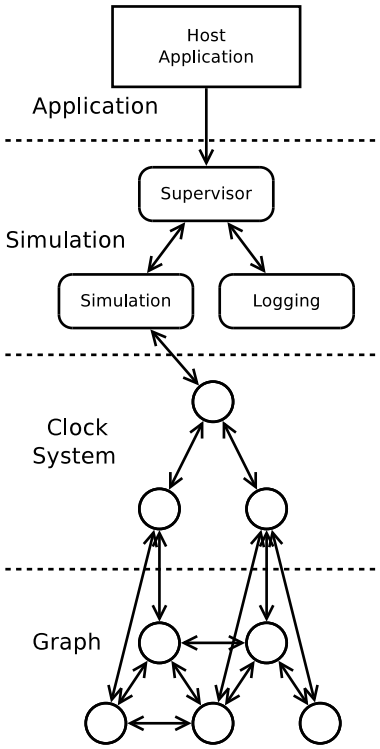

Figure 1: Overall architecture of the simulator

This allows us to have complete separation of behaviour of the nodes and the links. If links were not modelled as actors, messages would be passed directly from one node actor to another. While this is a standard pattern, there may also be other required behaviours of links. A user may wish to define a link type that randomly drops messages to model a non-perfect channel, or takes longer than a certain time frame to deliver messages to represent a link with a particular latency. Having links as separate actors allows us have this behaviour defined separately instead of having it mixed in with the code for the nodes.

Representing every single entity in the graph as an actor allows us to exploit the fact that within a single time instance, the entire graph is an embarrassingly parallel system. That is to say that all entities are able to execute without having to wait on other entities finishing their action first. This, coupled with an actor library that allows us to run multiple actors simultaneously on a certain number of threads creates a system that will scale well when the system resources are increased.

This idea of explicit links is shown in figure 2. In this diagram, squares represent hosts or simulation nodes, and circles represent the links between them. The nodes only communicate directly with the link actor representing the link to the node it wishes to contact. Nodes never contact each other directly.

3.3.2 The Clock Distribution

ScaNS is structured as a synchronous, tick-based simulator. This means that all simulation actors work simultaneously and they only perform their algorithm when instructed with a "Tick" message. This way I could easily control the concurrency of the system as tick based systems allow the

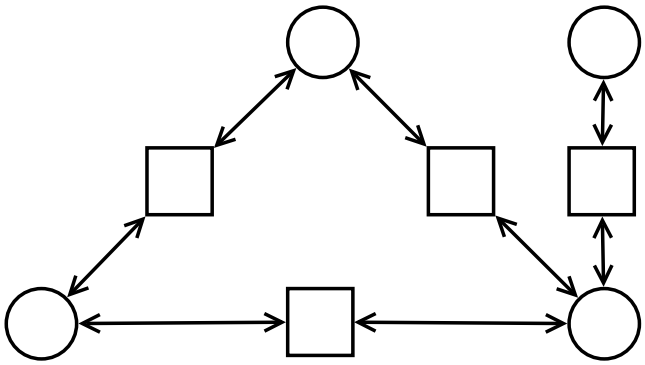

Figure 2: Showing the explicit nature of links in the system. Circles representing hosts and squares representing the links between them.

system to run indefinitely until all simulation objects have returned a "Tock" message to say they have completed their work for this time iteration.

The clock distribution system was designed in a way that I could multiplex signals that were required by all actors. Since the simulator is a tick-based simulator which requires a tick to be sent to signal the start of every time instance, and a response from every actor on every time instance, this was very important. The clocks serve dual purpose: sending messages from the simulation layer to all actors and retrieving logging messages from the graph objects to return to the simulation layer.

Initial testing showed that having a single clock entity that had to deal with signals and responses for every graph object created a bottleneck which caused a slow down of the system. This is understandable as a single actor is a single unit of concurrency so can only process ever receive message serially. A system needed to be created that would spread the work out over multiple actors, increasing parallelism.

The clock distribution system takes the form of a tree where there is a single clock at the top which takes commands from the simulation layer and there are several leaf clocks which then communicate directly with the graph objects. This allows us to spread out the work of the clocks over several leaf nodes, allowing us to increase the parallelism of the system as multiple clock signals can be processed at once.

Performance of the system is also increased through aggregation. Instead of immediately forwarding a message from the graph objects back up the tree, the clock will wait until it has received messages from all of it's children. It will then aggregate all of these messages in to a single object which is then passed up the tree. This removes the overhead of large amounts of unnecessary message passing.

3.3.3 The Simulation Layer

The simulation layer consists of three components: the supervisor, the simulator and the logger. The supervisor is the entry point for commands from the users host application. This creates a simple, single entry point to the simulator. It also functions as the management for the entire system, orchestrating the simulation and logger components.

The simulation layer is responsible for all aspects of the graph. It has the responsibilities of creating the actors that represent the graph from a given input, creating the clock distribution and then connecting the clocks to the graph. It is also responsible for maintaining state of the simulator to allow the serialisation of sent commands to it.

The logging component is the only way to get output from the simulator. The messages that were received at the supervisor are sent to the logger where they can be aggregated and finally output to a location and format defined by the particular logging component in use.

All three of these components are implemented using actors and all communication with them and between them are performed using message passing.

3.4 Host Application

The host application is the program which instantiates an instance of the simulator. The host application only needs to instantiate the supervisor and all other components are automatically generated. This makes the simulator simpler to use and doesn't clutter host application code, making it simple to integrate in to other applications. The host application must communicate with the supervisor through message passing as the supervisor is an actor, despite the host application not being an actor. The implication of this is that the simulator cannot directly pass data back to the host application as the supervisor doesn't know where to send the data as the host application has no mailbox. Therefore, the host application must rely on the logger component to get output from the system.

4. SYSTEM IMPLEMENTATION

This section will discuss the implementation of the ScaNS system. It will discuss low level ideas such as how I manage concurrency control for the system and how traits are used to create new simulation object types.

4.1 Simulation

The simulation component is the module in charge of the low-level maintenance of the simulation. It has the responsibility of creating and maintaining the simulation clocks, creating and maintaining the simulation objects and dictating the behaviour of how the clocks and simulation objects interact. The simulation component is structured as a finite state machine. This is only in effect during the initialisation phase of it's lifetime as it ensures that instance variables are instantiated in the correct order. Since during initialisation various components rely on other components having already been created, this can become an issue when objects are being created asynchronously using message passing. Using an FSM execution style allows a pipeline of initialisation to be performed where the next stage of initialisation is explicitly not started until the current stage has successfully completed. Standard execution involves simply flipping between the states of "working" and "ready". At the end of an working phase, as it's last action just after the component has moved to the ready stage, it sends a "Round Complete" message to the system supervisor. This message serves a few purposes. Firstly, it signals that this time iteration is complete and that the supervisor is clear to execute the next command in the future event list. This message also returns all logging messages generated by the simulation objects to the supervisor for processing, as well as a count of the total number of messages that were sent this time instance. This allows the supervisor to check for stable

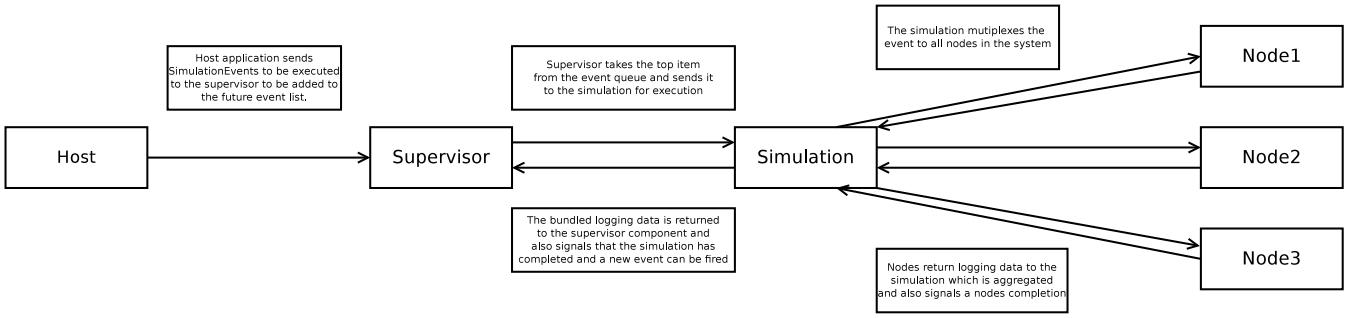

Figure 3: Message flow within the system

conditions.

The simulation component is used by the way of traits. A base trait for a simulation component is provided by the package which is then mixed in by a user class which fills in the abstract values. The most important function required is the "process" function. The base trait handles the reading of the input graph file in to a buffer and then calls the process function on each line in the buffer. By abstracting this function, I allow users to provide their own which will be able to handle the particular graph input format that is being used for this simulation. Example simulation components have been written to handle a subset of the graphviz DOT language [2], DIMACS format [1] and UCLA IRL format [3]. The user must also provide function for creating/removing a link and creating/removing a node. These functions are used for runtime modifications to the graph and allow the user to apply semantics to various operations such as defining the behaviour of the simulation when a link is removed.

4.2 Simulation Objects

The simulation objects of the system are the actors that represent nodes and links of the simulation. Simulation objects are explicitly split in to the two categories of "link" or "node". This allows us to enforce correct behaviour on how the simulator operates by always executing the the two in the same order. The correct behaviour of the simulator is that each time instance represents a single execution on all of the nodes, followed by a single execution of all of the links. This allows the simulator to perform deterministicly as messages will always arrive on the same tick when simulations are rerun. This is opposed to if this system was implemented with a unified simulation object type where every object in the simulation is executed simultaneously, leading to non-deterministic behaviour depending on which actor is executed first.

Simulation objects have two behaviour functions. A "per tick" function which is executed every time the object is ticked and a "per message" function which is executed on every message in the objects mailbox. This split allows the situation where the simulation must be started with no external inputs. An example of this is where the first iteration of an algorithm is to send it's state to all neighbours. This behaviour is not instigated by an incoming message and thus cannot be executed in the "per message" function. The per tick function also allows the used to output some state on every iteration of the simulation or perform an action on every tick. The per message function is executed for every message stored in an objects mailbox. Both of these

```
def Tick(){
  val perTickLogging = PerTickBehaviour()
  var perMessageLogging = ArrayBuffer[LogMessage]()
  for( msg <- messageBuffer ){
    perMessageLogging += nodeAlgorithm(msg)
  }
  val allLogging = perMessageLogging ++
      perTickLogging
  clock ! allLogging
}
```

Figure 4: The tick behaviour of a simulation object

```
def nodeAlgorithm(message){
  message.payload match{
    case x:TypeA => {...}
    case x:TypeB => {...}
    case _ => {...}
  }
}
```

Figure 5: The node algorithm matching message types

functions are specified by the user to embody the complete behaviour of a node. This behaviour is shown in figure 4. First the per tick behaviour is executed. This is executed once every time the node receives a tick representing a new time instance. The node algorithm is then called and executed on every message in it's message buffer. The node algorithm given in figure 5 shows the the typical structure given to the node algorithm. The type of the message that I am currently operating on is matched to a set of types that I have known actions for. This allows a node algorithm to handle various types of message such as handling both data and control messages.

The final function that must be provided by a new node type is "getState" which is executed in response to a stateRequest message from the simulation actor. This function takes a string argument, allows the user to optionally specify what state is returned by the object. This is a useful way of querying specific pieces of state of an object during runtime.

4.3 Supervisor

The supervisor component is responsible for the orchestration of the entire simulation from a high level. It receives commands from a host application via standard message passing and performs the command based on the message received. The supervisor maintains the future event list required by the simulator as well as acting as the main dispatcher for events within the system. This components execution is also modelled as a finite state machine. This allows the simulator to create a correct temporal ordering of events while making use of asynchronous message passing system. The supervisor maintains a state and the body of the supervisor defines what actions can be performed while in that state. Once an action has been performed by the supervisor, it's next action is to update it's state to an appropriate value as to block any other operation until the current one has completed.

As an example, when the supervisor is instructed to perform a logging operation, it first checks if it is in a ready state. If it is, then the supervisor can send logging information to the logging component and will then set it's state to "logging". While in the logging state, no other operations can be performed and the system will only exit the logging state when it has received a "logging completed" message from the logger. This will then reset the state of the system to ready and will dequeue the next event on the future event list.

The finite state machine of the supervisor combined with the future event list underpins the concurrency control of this simulator. This is a simple form of concurrency control that maintains correct state of the simulation by only allowing one operation to be performed at a time. This is in contrast to more complicated concurrency control methods which allow multiple operations to be performed simultaneously but require checks and rollbacks if an error is detected. For use by host applications, a class using the standard supervisor trait must be made to fill in three abstract values. These are a logging component, a simulation component and a function that will be called when the system is shutting down, allowing for custom shutdown behaviour such as dumping extra logging information.

4.4 logging

The logging component makes up the "view" portion of the MVC pattern. It's purpose is to receive messages generated by any other component and process them in some way. As there are many ways that a user may wish to generate output, a trait is used to provide the barebones of the logging construct and then all the user must provide is an "output" function which will take a message and generate some form of output. There are two standard use cases for the logging actor: Logging to StdOut and logging to a file. As such, these two primitive are provided by default. The logger component is general enough that it would be possible to write a process method that updates a GUI with data upon receiving each message as no return type is expected of the process function.

5. EVALUATION

Evaluations were performed on various aspects of the simulator to assess qualities such as scalability, memory usage and execution time. Assessments of the systems ability to scale with graph size is important as the Internet will continue to grow and the simulator must be able to handle larger datasets. Scalability with number of available cores, as well as overall memory usage and execution time metrics allow us to assess how the simulator would increase in performance when given resourceful machines and how performance is lowered when used on restricted resource machines.

```
def nodeAlgorithm(packet: SimMessage):SimReturn =
    {
  packet.payload match {
    case x: floodPacket => {
      var seen = seenBefore(x)
      if(!seen){
        for(interface<-interfaces){
          if (interface._1 != packet.source) {
            sentMessages += 1
            interface._2 ! SimMessage(id,
                interface._1, 0, packet.payload)
          }
        }
      }
    }
    case _ => {}
  }
}
```

Figure 6: The node algorithm for the flood type node

Evaluations were performed on a machine comprising of four 16-core AMD Opteron 6274 chips and 512GB of RAM running FreeBSD 9.1. For every evaluation the JVM was set to use the concurrent mark and sweep garbage collector, has an Xmx value of 200G and was executed using the "-server" flag. Basic variables that are tweaked in these evaluations are the input graph used, the number of ticks performed, the number of times the entire simulation is run without shutting down the JVM, or "rounds" and the number of messages that are injected in to the system before Tick one.

5.1 Scaling with Graph Size

The first evaluation was to assess how well the simulator performed in response to increasing graph sizes. The purpose of this evaluation was to see the effect on the simulator when running million-scale simulations vs smaller scale simulations. This is important as it would show both the range of potential applications for the simulator and will show that the simulator will still be of use in years to come when the size of the AS graph has continued to grow.

For this evaluation a "flood-type" node was used which sends a message it has just received on all of it's outputs except the one it was received on. This is shown in figure alg:flood. This allows us to generate a lot of traffic within the system very quickly and for a short period of time. Sequence numbers were used in the messages to avoid an exponential flooding of messages.

Ten ticks were performed as no changes occur after this point and the simulator would be simulating nothing happening. Only one injected message was used to start the simulator to first test on light work-load and heavy workload could be tested later. To test on graphs of increasing size, the AS graphs from 2004 to 2010 were used. This entire evaluation was performed 5 times to produce minimum,

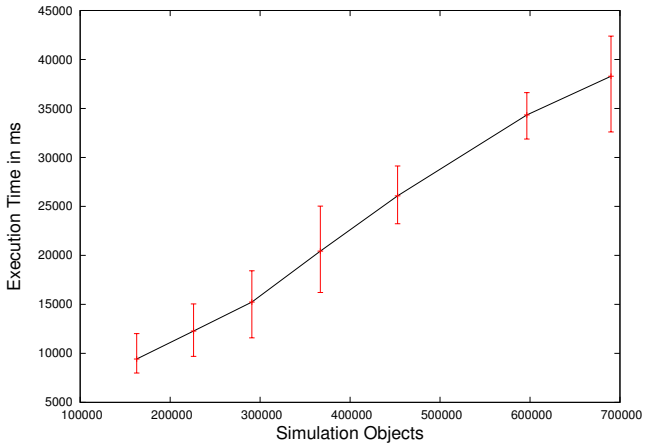

Figure 7: Execution time shown against the total number of simulation actors in the system using the IRL AS graphs

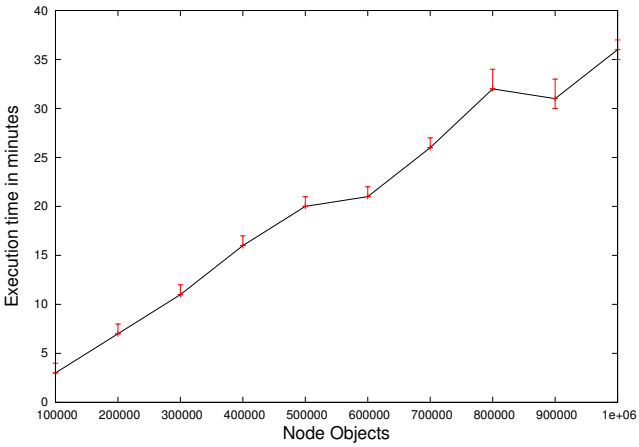

Figure 8: Showing execution time against graph size using the preferential attachment graphs

maximum and average values for execution times.

If the simulator is scaling well with respect to the size of the graph, I would expect to see a linear increase in the execution time with respect to the number of simulation objects in the system. This is due to the node algorithm taking being the most expensive operation defined by the system.

As shown in figure 7 the execution time indeed scales linearly with increasing size of the graph, showing that increasing the size of the graph will increase the overall execution time predictably.

As an extension of this evaluation, to test the execution time against even larger, Internet-like graphs, a set of 10 graphs ranging in size from 100,000 to 1,000,000 million nodes were created. These values were chosen as it was though that these values would be high enough to stress the system in to super-linear time growth. These graphs were created using the Barabási-Albert model which generates graphs using a preferential attachment mechanism as described in [6]. The purpose of this evaluation was to test how well the simulator deals with theoretical future AS graph sizes as well as other graphing problems that would require the simulation of several million entities. Although the number of nodes in these graph files seem reasonable for a large scale simulation, the number of the links in the system range from 1,403,922 for the 100,000 node graph to 13,998,616 for the 1,000,000 node graph. This creates a massive number of total simulation objects, ideal for stressing the system.

Expected results for this evaluation were that the simulator would continue to grow linearly in execution time with respect to the number of simulation objects in the system. Also expected was the towards the larger graphs I would start to see super-linear growth as the system begins to reach physical limits and exponential growth when it goes above these, tending to a graph size where simulation is time infeasible. Interestingly, as shown in figure 8, the simulator continues to have a linear increase in time with the number of simulation objects introduced. This suggests that the simulator has the ability to work well with even larger graphs than those used here as I have yet to reach the graph size of exponential time increase.

5.2 Messages effect on Simulator

The number of messages that will be sent during a simulation is entirely variable depending on what is being simulated. Some simulations may have only certain nodes responding to messages, keeping the total number of messages in the system quite low and others may have all nodes sending messages on every clock tick. To cover for the case where there are large numbers of messages in the system, I performed an evaluation to test the effect of the number of messages in the system against the execution time. This allows us to characterise how the system behaves under different stresses caused by fluctuating numbers of sent messages.

For this I perform all evaluations on the same graph was used, the 2008 AS graph. The evaluation was run for 10 ticks as no messages are sent after this point. Flood-type nodes were used and the number of injected messages were increased from 1 to 10. On the 10 injected messages point, 10 complete floods over the entire network will be performed simultaneously for different locations, creating large quantities of messages in the system and stressing the system. The ten inject points were chosen randomly from the input graph file. The flood type node is especially useful for this evaluation as it allows us to demonstrate the properties of messages within the system as these nodes have very little overhead in terms of execution time. Most of the execution time used by the system will be used for the transit of messages between nodes.

There are two parts to this evaluation: how the number of messages in the system effects the overall memory usage of the system and how the number of messages effects the execution time. Testing the memory usage allows us to characterise how much memory will be needed for a given simulation before it is run and testing the execution time will allow us to measure the overhead associated with message passing within the system.

As the number of injects to the system are increased, so do the total number of messages sent in the course of an entire simulation. Increasing from a single message inject to ten increases the number of messages in the system from around 2.27 million to 22.7 million, giving a large rage of

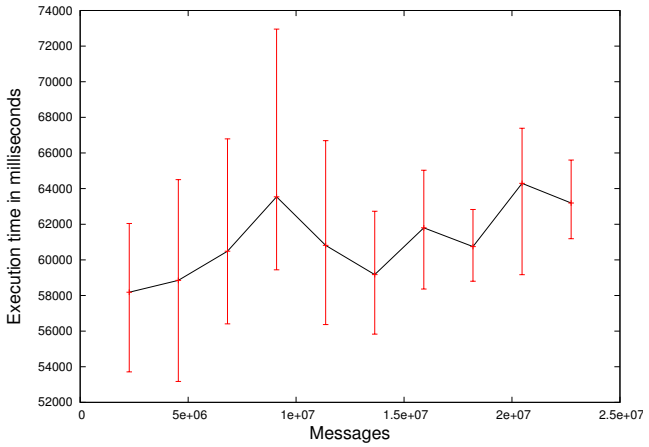

Figure 9: Execution time shown against the number of messages in the system

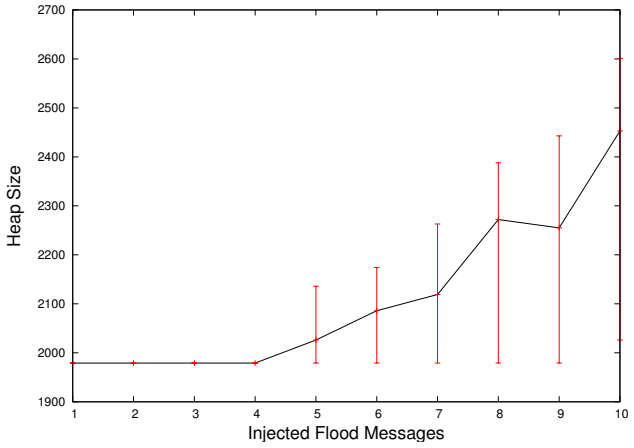

Figure 10: The size of the JVM heap against the number of flood messages injected

values to evaluate.

While evaluating execution time, it was expected that I would see an increase in the values with respect to the number of messages contained within the system. This is due to the overhead of each node having to process more messages and more links having to forward more messages. While evaluating the memory usage during the simulation, I would expect to see increases in heap size on iterations where exceptionally large numbers of messages are being send but for the memory usage to remain fairly constant.

Results shown in figure 9 show that execution time is independent of the number of messages within the system. This is shown by there being no discernible trend to the data and the large error bars showing that the range of values for all evaluations lie in a similar range. This would lead to the conclusion that there is no obvious statistical link between the number of messages in the system and the execution time.

Results shown in 10 show that memory usage behaves as I had predicted. On simulation ticks where exceptionally large numbers of messages are sent, the size of the heap is increased slightly. The area of the chart showing that between 1 and 4 injected messages the heap size is constant, shows the initial size of the heap as for this number of messages the heap size does not need increased. It can also be seen that the increase in heap size from the 2.3 million message run to the 22.7 million message run is rather minimal considering the large jump in messages. I only see an increase in the final heap size from about 1.9GB to 2.4GB. This again highlights that the messages within the system is of little consequence to performance of the system as the majority of this heap is used to store data concerning the simulated graph, which will increase linearly with the size of the graph.

5.3 CPU Resource

In order for the system to scale well with the continued increase of processing power, it was important to test how well the simulator handles increases in current processing power. This will enable us to see if, as the number of cores on a chip increase and number of chips on a board increase, I will continue to have an increase in speedup of the system. To do this, I tested how the simulator performed in constrained CPU environments. This would simulate the increase from a small number of cores machine to a larger, server-class machine.

To perform this evaluation, the cpuset utility was used to constrain what processors the JVM was allowed access to. This removed the need to use machines will less physical cores. The simulator was executed using the same AS graph for every iteration. The simulator performed ten ticks as no messages are sent after this point and the simulator would be simulating nothing. Ten message injects were performed at the start of the simulation in an attempt to stress the system and hence, achieve a high requirement for parallelism.

As shows in figure 11, the application exhibits super-linear speedup at 2 cores used and then continues to increase to 30 cores where where no extra speedup is achieved. Interestingly, the trend line shows speedup being reduced when the system is given between 40 and 64 cores. There are a few factors that could contribute to this result. It may be that the system was not stressed enough to require the use of so many cores and telling it to use so many cores was creating large amounts of thread switching in the actor dispatcher. Another possible explanation is that it is an artefact of the type of simulation run and that a different simulation type would yield a different speedup profile.

6. APPLICATIONS

To prove the worth of a system, it is required to show that the system actually has potential applications. Here I show ScaNS for use with evaluation of a particular algorithm: The distributed K-core algorithm.

6.1 K-core Decomposition

To test that the simulator was fit to simulate the sort of graph routing problems that I am targeting, a simulation of the distributed K-core algorithm was create. This also demonstrates that the simulator will work for real applications, as well as the synthetic flooding examples used for the other evaluations.

A version of the distributed K-core algorithm defined in [12] was written for use with this simulator. As the algorithm already existed in Java, the translation to Scala was purely mechanical. The resultant code was very similar to

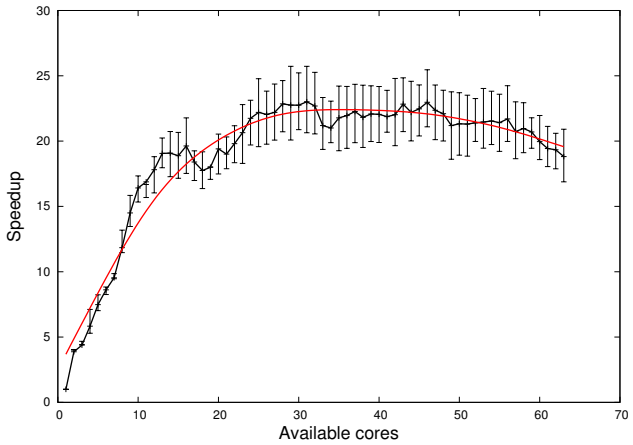

Figure 11: Speedup shown in relation to the number of available CPU cores

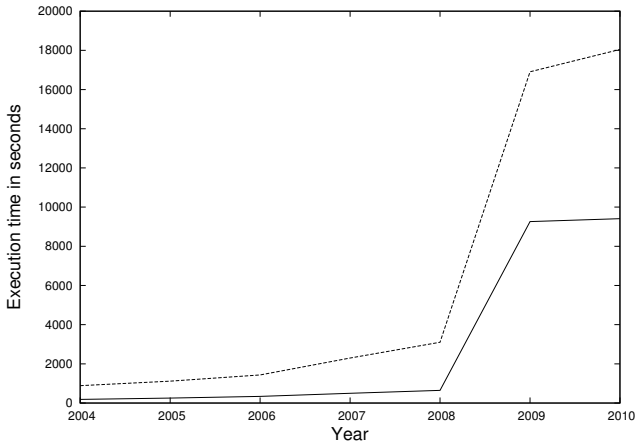

Figure 12: The execution time to convergence for ScaNS(lower line) and the custom K-core simulator(upper line)

the original code as code translated from java to scala will tend to be. The algorithm implementations have identical flow control and the only changes were small changes to make the algorithm communicate with the new simulator instead of the old simulator. The application was executed on the AS graphs for 25th of October on both ScaNS and on an existing simulator purpose built for running this algorithm but only runs in a single thread.

6.1.1 AS graph evaluation

Expected results from this evaluation were that ScaNS would outperform the bespoke simulator substantially, despite being a general purpose simulator.

Figure 12 shows that ScaNS has a consistently lower execution time and also has slower increase in the time taken to convergence with the size of the graph. This is explained by the custom simulator being single threaded and ScaNS using actor parallelism. The slower increase can be explained by the simulator not having reached a point of saturation that would cause time to increase.

The large increase in execution time shown between the

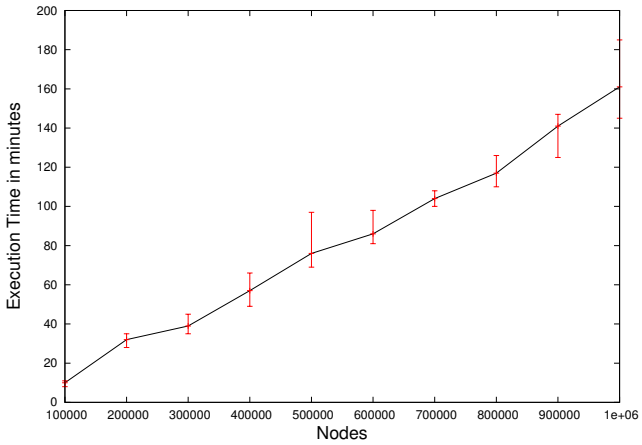

Figure 13: The execution time to convergence on preferential attachment generated graphs

years of 2008 and 2009 appear to be the result of a large increase in links and several nodes that now have a very large degree. This is potentially a glitch in the data for these years or may be due to a new way that the topology was collected, resulting in a more accurate picture containing more links. The first explanation is more plausible as the data returns to reasonable values after the year 2010. However, as this is a glitch with the dataset which shows up on both simulators, the results can still be counted as valid.

6.1.2 Large graph evaluation

To further test the simulator with distributed K-core decomposition, an evaluation was run using the 100,000 to 1,000,000 node preferential attachment graphs to see how well the simulator deals with both a realistic application and a very large graph. This is an extension to the last evaluation as it shows that even as the size of the AS graph continues to grow, the simulator will still function and continue to be useful.

Expectations for this evaluation were that the execution time would continue to increase linearly with the size of the graph and at a certain saturation point the execution time would begin to increase super-linearly.

Interestingly, figure 13 shows that the simulator continues to exhibit a linear increase in execution with graph size. This tells us two things: that I have still not reached a saturation point with respect to the graph size where I begin to see super-linear behaviour and that even for real world applications, ScaNS still executes in a reasonable time.

6.2 Discussion

This section evaluated ScaNS against various metric to determine it's fit for purpose. There are several conclusions that can be drawn from these results. First is that the execution time of the simulator increases linearly with the size of the graph, adding a predictable quality to the simulator and showing that simulations will take reasonable time with even very large data sets. Next was that the overhead of nodes passing messages within the system is very minimal. The actions that are executed in response to the messages may be substantial but the passing and processing of the messages themselves has minimal impact on execu-

tion time and memory footprint of the system. The evaluation of execution time against number of available cores presented interesting findings. While it can be seen that the system will speed up substantially when given extra cores to use up to around 32 cores, where it was expected that speedup would remain constant at a point, speedup actually begins to decrease when given more cores. Finally, a real-world application of K-core decomposition was given. This evaluation showed that ScaNS, which is designed to be a general simulator, outperforms a custom simulator built for this application. This shows that ScaNS performs well for very specific application even though it was deigned to be general enough for many uses.

7. RELATED WORK

There are two main bodies of work on which this paper is based: Simulators and data sets. In this section I will present important examples of both which had an impact on the development of this system.

7.1 Simulators

There are currently many packet-level simulators available. NS3 is a packet-level simulator aimed at research purposes. NS3 has a large degree of flexibility in that many aspects of simulations can be customised. These include being able to specify network stacks and being able to install "applications" on particular nodes. It would be technically possible to perform high-level simulations using this tool but they would become impractical due to the number of nodes that need to be defined. Also, at it's core, NS3 is a single threaded application. This would cause simulations of the size I am studying to run slowly even if the computer had extra available compute resources.

GTNetS [15] is another packet-level simulator but is closer to our goal. This simulator is primarily for use with moderate to large scale networks. It simulates fine-grained elements of the network, such as network stacks, just as NS3 does. However, GTNetS achieves it's ability to support larger networks as it is a multi-threaded application. Again, a simulator which allows this fine-grained simulation is not required for our class of applications and would lead to unneeded code and potential over-verbosity of the output from the simulator.

An example of a tick-based discrete event simulator that provided the base ideas for ScaNS is presented in "Programming in Scala" [13]. The authors present a tick-based simulator which is used to simulate digital logic circuits in parallel. The main idea taken from this is that all entities in the modelled logic circuit are represented as scala actors. It also emulates this example in the way that the ticks are distributed from a clock component which is also represented by an actor.

7.2 Data Sets

The graph input to the simulator is one of the most important elements. There are a number of sources of AS graph data all with pros and cons for various applications.

The Internet Research Lab (IRL) Topology data set [18] from UCLA was the main data set used in evaluations in this paper. This was due to it's simple tab-delimited structure being very easy to parse. The IRL topology project also offer a "links" data set that removes a lot of the unneeded data from the total data set and only represents the links between ASes. As this is exactly the data that I required and no more, this was the data set that was used.

Another popular source of AS graph topology data is the Route Views project [4]. This project maintain machines around the world that collect BGP data that can be used in research. While using BGP data will give an accurate image of the AS graph, using routeviews also has some disadvantages. The data available is low level BGP data which must be processed to yield a data set representing the AS graph. Also, to get the most accurate graph, the data must be downloaded from all collector nodes around the world, merged and processed in order to create a more complete graph and remove duplicates.

8. CONCLUSIONS AND FUTURE WORK

There are two main areas I would wish to look at for future work: The finite state machine and the evaluations.

ScaNS is implemented using a finite state machine that was written from scratch. However, Akka offers a finite state machine type simply called "FSM". I would be interested in looking in to if the custom implementation works more efficiently as it was written specifically for this simulator, instead of being general purpose like FSM. It may also be the case where FSM performs better as it has a more sophisticated system underneath. If it was found that FSM works better with scans then the major components would need to be rewritten as FSM uses a domain specific language to define finite state machines.

All of the evaluations presented in this paper are examples of a distributed algorithm running on a large network where all nodes are running the same algorithm. This is not representative of a real world network where many devices are running different algorithms. An example of this is that host PCs will be running a different algorithm to a router. While it is possible for the graph to be represented as several different distributed algorithms, this was not tested during the evaluation. This was due to focus on specific applications where every node does run the same algorithm. This evaluation would be especially interesting in the case where some nodes have algorithms that take substantially longer than others. I would hypothesise that the synchronous nature of ScaNS would cause the execution time to grow faster the more of these slow nodes are in the graph.

Finally, all evaluations shown in this paper are example of algorithms being executed on static graphs. That is to say that the graph is instantiated once at start up and it not modified for the entire length of the evaluation. This also presents an unrealistic scenario as devices are often connected and disconnected from the network. ScaNS does have the functionality to remove and add nodes and edges at runtime although this was never evaluated. This is the next logical step for the K-core application as this algorithm exhibits different behaviour when a new node is added to the graph than when the entire algorithm in initialised.

In this paper I presented ScaNS, the first scalable high-level routing simulator to use the actors model of concurrency. ScaNS models everything as an actor in an attempt to use as much of the computing resources available to it. I have provided evaluations which demonstrate it's scalability in respect to the input graph sizes, as well as demonstrating the speedup available when moving to multi-core, multi-processor machines. I have also demonstrated ScaNS fit-for-purpose through the use case of a distributed algorithm

for use with next generation routing, K-core decomposition. The final evaluation also demonstrated that the simulator will still execute in reasonable time, even when presented with very large graphs.

This simulator represents a new class of distributed algorithm simulator where actors are used to directly model graphing problems and this paper presents the case for this model.

9. REFERENCES

[1] Dimacs implementation challenges, http://dimacs.rutgers.edu/challenges/.

[2] The dot language, http://www.graphviz.org/doc/info/lang.html.

[3] Internet topology collection file format, http://irl.cs.ucla.edu/topology/#format.

[4] ANTC Oregon. RouteViews.

[5] J. Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, KTH, 2003.

[6] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *science*, 286(5439):509–512, 1999.

[7] T. Bu, L. Gao, and D. Towsley. On characterizing BGP routing table growth. *Computer Networks*, 45(1):45–54, May 2004.

[8] S. Carmi, S. Havlin, S. Kirkpatrick, Y. Shavitt, and E. Shir. A model of internet topology using k-shell decomposition. *Proceedings of the National Academy of Sciences*, 104(27):11150–11154, 2007.

[9] K. F. D. Meyer, L. Zhang. RFC 4984.

[10] C. Hewitt, P. Bishop, and R. Steiger. A universal modular ACTOR formalism. In *Proceedings of the International Joint Conference on Artificial Intelligence*, Stanford, CA, USA, August 1973. Morgan Kaufmann.

[11] G. Huston. Analyzing the Internet's BGP routing table. *The Internet Protocol Journal*, (July):1–16, 2001.

[12] P. Jakma. Distributed k-core decomposition of dynamic graphs. *ACM CoNEXT*, 2012.

[13] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima Incorporated, 2008.

[14] Y. Rekhter and T. Li. RFC 1771. *http://tools.ietf.org/html/rfc1771.html*, 1995.

[15] G. Riley. The georgia tech network simulator. *on Models, methods and tools for reproducible network*, (August):5–12, 2003.

[16] S. D. Strowes and C. Perkins. Harnessing internet topological stability in thorup-zwick compact routing. In *INFOCOM, 2012 Proceedings IEEE*, pages 2551–2555. IEEE, 2012.

[17] M. Thorup and U. Zwick. Compact routing schemes. In *Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*, pages 1–10. ACM, 2001.

[18] B. Zhang, R. Liu, D. Massey, and L. Zhang. Collecting the Internet AS-level topology. *ACM SIGCOMM Computer Communication Review*, 35(1):53–62, 2005.