Modelling and Simulation of Aggregation Nets*

Alex Poylisher
Department of Computer Science
University of Warwick
Coventry CV4 7AL, UK
sher@dcs.warwick.ac.uk

Michael Luck
Department of Electronics and Computer Science
University of Southampton
Southampton SO17 1BJ, UK
mml@ecs.soton.ac.uk

Abstract

In large-scale service monitoring, automated dynamic (re)distribution of running monitoring applications is likely to push the limits of scalability. In the aggregation nets *approach, we associate each partition of a distributed application with an autonomous agent capable of relocating the partition fully or partially, or modifying it to accommodate the dynamics of its local environment. Coordinated agent behaviour is to result in maintaining acceptable performance for the whole application. Aggregation nets is a typical Grid application that relies on the availability of distributed computing power and a network sensing infrastructure that provides information for agent decision-making.*

In order to evaluate our approach, we emulate aggregation nets on top the physical network simulator, Berkeley ns. *This paper describes our model of aggregation nets, and its implementation.*

1. Introduction

Service monitoring applications continuously update aggregated and filtered views of raw management data relevant to the performance of the monitored service. Since this raw data is extracted from the distributed hardware and software components of the service, questions of scalability arise. Indeed, it has been convincingly argued (e.g., [6]) that scalability of monitoring applications in large networks and with complex services can be achieved only by distributing the application so that raw data is not transferred over long network distances and processing is distributed among multiple processors.

Distributed monitoring systems built over the last decade are hierarchical in nature, with a manual distribution of computations that does not change at run time. Such systems are thus still limited in terms of scalability in large and dynamic environments, in which automated and dynamic (re)distribution of a running monitoring application may help to push the limits further. Here, it is impossible to obtain an accurate snapshot of the whole network in a single location in time to make meaningful (re)distribution decisions. Consequently, a decentralised (re)distribution approach is called for; our work is directed towards applying network-aware mobile agents to manage re-distribution of service monitoring applications at run time.

In [10] we provided an analysis of the limitations of current distributed monitoring systems and introduced a possible agent-based solution, *aggregation nets*. The basic idea is to associate each partition of a distributed monitoring application with an autonomous agent capable of relocating the partition fully or partially, or modifying it to accommodate the dynamics of the environment, including fluctuations of resource availability (effective bandwidth, latency, host load) and changes in the data source (service component) or consumer (human or automated manager) populations. Agent behaviour is guided by its own analysis of the local network environment (including predictions of the future), relationships of the partition it carries with other partitions, and histories of past decisions. Maintaining an acceptable performance level of the application as a whole, through its lifetime, can thus be achieved by coordinated local agent decision-making.

Aggregation nets rely on the availability of distributed computing power and the network sensing infrastructure that can support agent decision-making. In our view, this makes aggregation nets a typical Grid application, whose *design* becomes simpler with the assumption of Grid services, and can influence the *requirements* for such services.

In order to evaluate our approach, we emulate aggregation nets on top of a publicly available physical network simulator, *ns*[3], which provides a controlled environment for aggregation nets to be studied at a scale that is practically impossible on a real network. This is the necessary *initial* step in building an instrumented micro-world for experimentation with different agent-based solutions.

*This project is funded in part by Telcordia Technologies.

This paper describes the basis for our approach, and details the simulation environment and its construction. We concentrate on providing a complete rationale and description of our tools, and leave the discussion of the experimentation to a subsequent paper. Section 2 reviews the concept of aggregation nets, Section 3 describes our model of aggregation nets, Section 4 briefly discusses implementation of the model on top of *ns*, and Section 5 summarises our experience.

2. Aggregation nets

We define an *aggregation net* as a partitioned monitoring application whose partitions are embodied in autonomous agents (known as *drifters*), which execute only in *execution environments* (EEs) that provide services to them. These drifters are physical, network-aware and self-observing agents capable of mobility, shifting a part of the partition they carry to other drifters, and modifying the partition (drifter capabilities reflect the possible operations on a running program). Associating the point of control over a partition's location with the partition itself (as opposed to the host machine) allows us to keep some history of prior re-mapping and re-partitioning decisions that can be used by an agent in more effective future decision-making. In our approach, coordinated local actions of drifters are to result, most of the time, in desirable global behaviour for the application as a whole.

As the execution of an application proceeds, its performance is affected by the dynamic aspects of its environment. These include *network dynamics*, such as fluctuations in connectivity, available bandwidth, latency, and processor loads, and *population dynamics*, such as changes in the number and location of data sources and consumers in the course of the computation.

Specifically, a monitoring application must satisfy the constraint that it obtains input data at specified polling intervals (to ensure correctness of results, regardless of delivery time to consumers); it must achieve the objective of striving to deliver results to consumers with specified periodicity; and, with in-band monitoring, it must achieve the objective of minimising its own intrusive effects on the resources it shares with service-related traffic and computations.

Regardless of the precise definitions of these performance criteria, local drifter decision-making is based on three kinds of information: data on current and projected degrees of utilisation for the resources of interest to the drifter (including current/projected effective bandwidth/latency between selected network nodes, and current loads on a set of EEs); self-observation data on the way current resources have been used by the drifter in the near past; and information on the proposed actions of other drifters in some portion of the aggregation net (most importantly, neighbours), for coordination of actions.

The data of the first kind requires a network sensing infrastructure external to drifters or EEs. An aggregation net will make use of a network of distributed monitors along the lines of the Network Weather Service [11, 13], which has its own cost of operation, in terms of resources used and adverse impact on the service. Though a full evaluation of the benefits of aggregation networks must account for this cost, if such an infrastructure is used by more than just monitoring applications, as would be the case in the Grid world, then the cost becomes smaller, and can be disregarded if the infrastructure is a standard part of wide area networks.

The goal of the aggregation nets project is to develop a drifter agent architecture that would, at run time, result in organising drifter behaviour locally so that the application as a whole maintains its performance according to the chosen criteria. Given that drifters are situated agents, our first step is to understand the micro-world of the individual drifter. To experiment with different alternative architectures, we need an environment in which this micro-world is maximally realistic, yet sequences of events can be replayed, different physical network topologies and technologies used, experiments can be conducted on the appropriately large networks and applications, and statistics of interest can be collected. A network simulator provides us with a basis for such an environment. The rest of the paper describes our working model of aggregation nets and its implementation in *ns*.

3. Modelling aggregation nets

Operator nets The task of automated distribution requires a monitoring application to be represented in a form that is easy to (re)partition. Dataflow is one model of computation that both fits very well with the user model of a monitoring application as a continuously operating network of filters [5] and allows easy partitioning and distribution of a running computation. The operator nets formalism [1], in which an expression of such an application is functionally equivalent to a program in a dialect of the intensional language Lucid [2], fits these requirements well. Here, operators act on sequences of values, in one or more dimensions.

A simple operator nets program that computes a running average of the values of two variables in one dimension (time) is shown in Figure 1. At a high level of description, the computation proceeds as sequences of values "flow" from the input vertices i_1 and i_2 through a network of arithmetic ($+$, $/$), constant (1, 2), intensional operator (**fby**, **next**) vertices to produce a sequence of averages at the output vertex *o*. A black circle indicates a so-called **split** vertex used to create two identical copies of the incoming

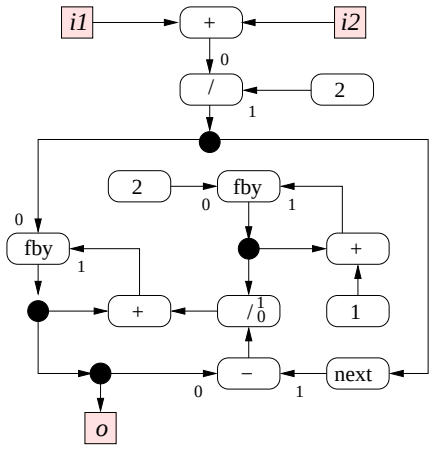

Figure 1. Operator net for Running-Average-2

value. Where order of the incoming edges for a vertex is significant, the edges are numbered.

If results of partial computations are labelled a, b, and c, then the following equations specify the mathematical semantics of the program (for a detailed description of semantics of intensional operators refer to [1]):

$$a_t = \frac{i_{1_t} + i_{2_t}}{2}; b_t = \sum_0^t a_t; c_t = \sum_0^t 1; o_t = \frac{b_t}{c_t}$$

In a monitoring application, the input variables may be the values of two Simple Network Management Protocol (SNMP) Management Information Base (MIB) [4] variables with identical object identifiers on different hardware components. Note that the program as shown has exactly two inputs, and that number was known at compile time. This example demonstrates simultaneous spatial (two input streams that can be derived from two distinct data sources) and temporal (running average) aggregation.

As monitoring applications are long-running processes, we need to express aggregation over a set of inputs such that membership in the set can change at run time, and have the application itself adjust to any changes. In our example, we would like to specify that a running average of the values of N identical variables (derived from distinct data sources) needs to be computed, where N is unknown at compile time. In general, the set of variables over which some aggregation computation is defined need not be homogenous, and the definition of the computation is separate from the definition of the set.

Multidimensional versions of Lucid (and the corresponding operator nets) can deal with computation over sequences in an unlimited number of dimensions. Hence, by switching to another dimension (for instance, the dimension of IP addresses) it is possible to express a computation over a finite set of values corresponding, for instance, to the same time and different location, without representing the number of elements in the set explicitly in the program. Details are beyond the scope of this paper, but we note for the following discussion that run-time support for such flexible sequences is one of the design requirements.

To be executed on distributed processors, a program such as the one in our example needs to be divided into more than one partition, and the partitions must be mapped to processors. Input and output vertices must be connected to the appropriate data sources and result consumers. A possible 3-way partitioning and mapping for the program in Figure 1 is shown in Figure 2. A mapped aggregation net derives input data from *data sources* (SNMP 'agents') and provides results of the computation to *consumers* (clients outside the aggregation net). Initial graph partitioning and partition assignment problems are difficult, but a number of solutions exist (see [12]), and we assume that a good initial partitioning and mapping can be found. Because monitoring computations are long-running, we focus on their execution following the initial mapping and startup phase.

In order to maintain acceptable performance in the face of changes in the environment three operations are theoretically possible on a running operators net program. These include *re-mapping* whereby a partition is moved to another machine, *re-partitioning* whereby a part of a partition is moved to another partition (possibly on a different machine), and *modification* of the operator nets program itself to an equivalent program (using the properties of operators such as the associative law of addition).

A model for operator nets In modelling aggregation nets, we aim to strike a balance between abstracting inessential detail, and keeping the model grounded so that the architectures we prototype can be easily transferred to existing or near-future implementation platforms. An operator net model should be sufficiently general to represent a wide range of real or synthetic aggregation applications, yet avoid unnecessary complexity of a complete implementation. Our model is based on *flat operator nets* [7], a programming model derived by restricting the syntax of general operator nets, to consist of a main net (digraph) of vertices and edges, and a set of subsidiary function nets for each function referred to in the main net. Subsidiary function nets cannot themselves have subsidiary function nets (functions are not allowed to be defined within functions), and a flat operator nets program cannot have any subcomputation vertices.

We need to model a small set of operators from the operator nets repertoire that could be used to define all operators in a given algebra (as functions). According to [1], for the Lucid algebra this set includes a variety of **pointwise operators** (of different arity, such as the binary '+', an unary '-'

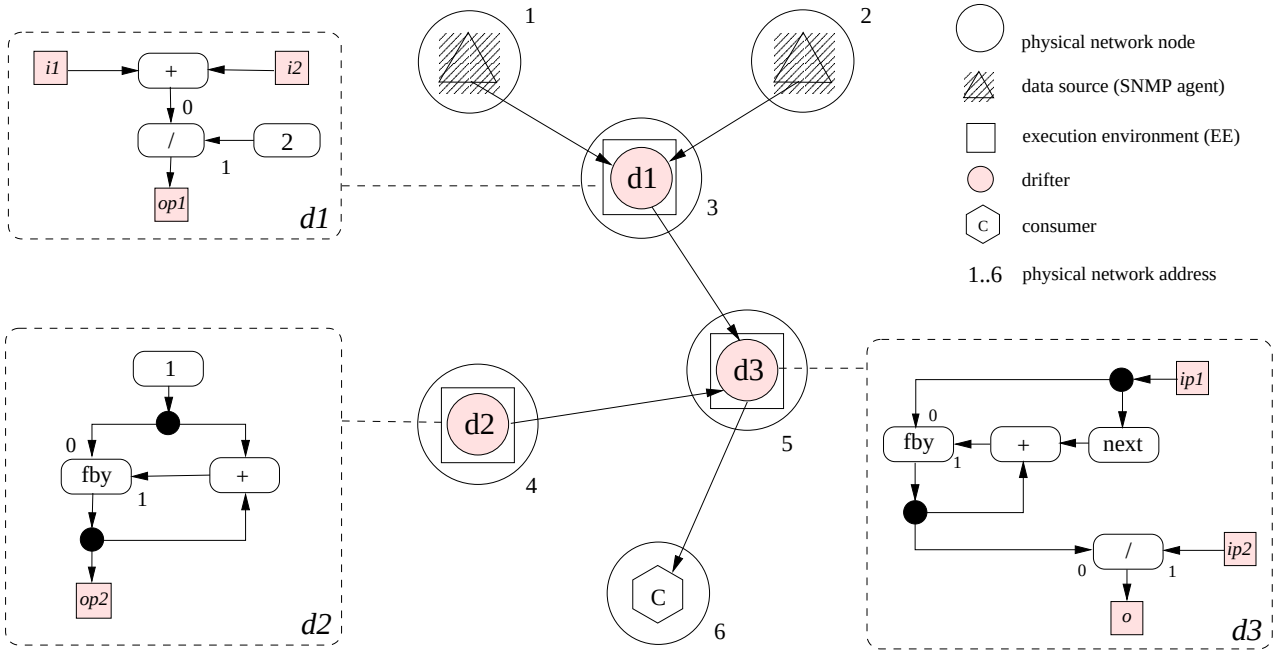

Figure 2. A partitioned and mapped operator net for Running-Average-2

etc.), operators that produce **constants**, the pointwise operator **if-then-else**, and the non-pointwise intensional operators **next** and **fby**. Apart from the operator vertices, we model **constant**, **split**, **input** and **output**, and **input port** and **output port** vertices. Only our application-specific extensions and restrictions are described below, with the rest of the vertices modelled according to [1].

We assume a hybrid, *eazyflow* model of evaluation for operator nets programs, wherein two kinds of dataflow 'particles' are placed on and removed from the edges of the network as it is evaluated. *Questons* embody requests for values and are placed by vertices on their incoming edges, and *datons* that carry values and are placed by vertices on their outgoing edges. We call queston production *backfiring*, in contrast with *firing* for daton production.

For all vertices, we are interested in the time it takes for a vertex to fire and backfire, given that the conditions for firing and backfiring are satisfied, *and* given the current load on the machine where the vertex code executes. There are two *generic attributes* that reflect vertex type and are used at run time to compute the simulated time to fire and to backfire. In the following discussion, we assume that a vertex fires or backfires after an appropriate delay, dependent on the current machine load.

An **input vertex**, responsible for polling external data sources for individual variables, is described as follows:

- edges: one outgoing,
- attributes: physical network address of data source, variable identifier, polling frequency, timeout value for requests,
- backfires: at polling frequency,
- fires: when a response arrives from the data source, by placing datons on the outgoing edge. Daton tags contain simple sequence numbers. Daton value type corresponds to the variable type.

An **output vertex** is responsible for keeping track of the consumer, generating questons inside the network at the frequency specified by the consumer, and sending results to the consumer. It is described as follows:

- edges: one incoming,
- attributes: consumer address and frequency for generating questons,
- backfires: at the frequency specified by the consumer,
- fires: when a daton is placed on the incoming edge.

The tags on the generated questons contain simple sequence numbers. The firing initiates transfer of the value of the daton and its tag to the consumer.

An **input port** vertex is used to receive datons into a partition of an operator net from other partitions, and to send questons to other partitions, and is described as follows:

- edges: one outgoing,
- attributes: id of the counterpart output port, id of the counterpart partition,
- backfires: when a queston is placed on the outgoing edge, by sending a request for a queston to be generated by the counterpart port,
- fires: when a request to generate a daton arrives from

the counterpart port, by replacing the queston with the tag in the request on the outgoing edge with a new daton with the same tag and the value in the request.

An **output port** vertex is used to receive questons into a partition and to send datons to other partitions, and is described as follows:

- edges: one incoming,
- attributes: id of the counterpart input port, id of counterpart partition,
- backfires: when a request for queston to be generated arrives from the counterpart port, by generating a queston with the tag specified in the request and placing it on the incoming edge,
- fires: when a daton is placed on the incoming edge, by sending a request to generate an identical daton to the counterpart port. When the request is received by the counterpart, daton is removed from the incoming edge.

Note the correspondence between input and output ports (labelled *ip* and *op*) in Figure 2.

A **generalised pointwise operator** vertex emulates any pointwise operator with the exception of **if-then-else**, and is described as follows:

- edges: number of incoming edges equals operator arity, one outgoing. The order of incoming edges is not important, as we are not interested in the result,
- attributes: generic,
- fires and backfires: as in [1].

An edge is characterised by *width*, which is the size (in bytes) of the daton values stored on the edge. For simplicity, we model only basic integer data types: a byte, an integer of 4 bytes, and a long of 8 bytes. An *edge* has two fixed-size sets associated with it: the queston and daton *stores*. In our model, the size of the stores are the same. An edge is labelled *eager* or *lazy*. If an outgoing edge of a vertex is labelled as eager, the vertex must fire whenever there are datons with some tag present on the vertex's incoming edges. If an outgoing edge is labelled as lazy, the vertex does not fire even if there are datons with the same tag on all incoming edges, unless there is also a queston with the same tag present on the outgoing edge. An algorithm that classifies edges into the two classes is described in [7].

In case a store is full and a new 'particle' is placed on the edge, a disposal discipline is required to remove 'particles' that have been un-consumed for too long. The discipline is an edge attribute, and the default is a simple *retirement scheme*, whereby if a 'particle' has not been demanded for some time (measured in terms of the number of garbage collections it survived), it is retired at the next collection. A more sophisticated scheme (not implemented), hinted at in [1] is based on *future usage count* and is not heuristic.

Data sources and consumers

A data source models an SNMP server ('agent') and implements a single SNMP operation. Upon receipt of a **get** request, which includes a variable identifier and the address of the requester, a data source returns the value of the variable to the requester, using a hash table. There is also a service operation to import a new variable, or change the value of an existing one.

Life-cycle	Inter-drifter	Computation control
on-creation	send-queston-request	suspend-firing
on-dispatch	rcv-queston-request	resume-firing
on-arrival	send-daton-request	suspend-backfiring
on-disposing	rcv-daton-request	resume-backfiring
run	send-subnet	start-generate-requests
split-subnet	rcv-subnet	suspend-generate-requests
		resume-generate-requests

Drifter-data source	Drifter-consumer
request-start-polling	subscribe-consumer
request-stop-polling	unsubscribe-consumer
	send-result-to-consumer

Table 1. Basic drifter API

Conversely, a consumer is a sink for the results of a continuous aggregation computation but, as we are not interested in results as such, it is modelled only for simulation of network traffic and debugging. It implements three operations: **send-subscribe-request** and **send-unsubscribe-request** to start and stop generation of questons inside an aggregation network, and **receive-result-from-drifter** to handle result arrival events.

Drifters Basic drifter functionality is summarised in Table 1, but does not include decision-making or inter-drifter coordination mechanisms still under development. The drifter life-cycle is based on the assumption of weak mobility, in which agents are responsible for transfer of their own state, and implement handlers for predefined life-cycle events that are executed in fixed sequence by host EEs. Drifters are modelled after IBM's *Aglets* [9]. The basic drifter state includes the full state of its partition (graph representation and edge content), scheduled firings and backfirings, a list of addresses of neighbour drifters, the timestamp of the last dispatch, and addresses of the current and last EE. When decision-making and coordination capabilities are added, drifter state will be expanded accordingly.

New drifters can be created in the same host EE as an existing drifters by the **split-subnet** operation, which transfers part of the existing drifter's partition (or subnet) to the new drifter. (A subnet is described by a list of vertices, so that splitting a subnet involves the creation of new input/output ports and possibly split vertices.)

Dispatching a drifter to a new EE is an EE operation. Data source polling is performed by the host EE on behalf of drifters and drifters need only to initiate, stop, or suspend polling. Basic inter-drifter communication involves pass-

ing questons and datons between input and output ports of neighbouring drifters, and transferring a subnet to an existing drifter. Computation control operations are used to pacify/restart computation inside the drifter partition, before splitting/transferring subnets or dispatching the drifter to a new EE. There are three kinds of events to manage: queston generation by the output vertices (if any), backfirings, and firings.

The life-cycle can be described as follows. When a drifter is created in an EE (initial mapping), its **on-creation** handler is executed. If its partition contains input vertices, the handler requests the host EE to start polling data sources; if it contains output vertices, the drifter starts generating questons (assuming that a consumer's address is stored in the output vertex before initial placement).

The remaining stationary behaviour is driven by the events of arrival of polled values, and queston and daton requests from other drifters. These are passed to the appropriate input vertices and/or input/output ports, triggering the firing/backfiring of these vertices which, in turn, trigger execution of internal vertices.

When a drifter decides to migrate, split a subnet into a new drifter, or transfer a subnet to an existing drifter, execution of its partition must be suspended, and resumed after migration/split/transfer. In a real mobile agent platform, each vertex has its own thread of execution, but in our model, we explicitly schedule firing and backfiring events for each vertex. Similarly, rather than simply restarting threads, we re-schedule all suspended events once the migration/split/transfer is complete, maintaining the order and intervals between original events.

The correctness of computations performed by aggregation nets depends on the accurate acquisition of polled values. Polls should be initiated with fixed frequency, and results should not be missed because of drifter migration or suspension. Delegating polling to EEs allows local and remote caching of polled values while the drifter on whose behalf the polling occurs is suspended. This means that a drifter can ask a remote EE to start polling on its behalf in anticipation of a future move or subnet transfer. The polled values are cached at the remote EE until the drifter/subnet migrates to that EE or remotely cancels polling on its behalf already in progress. When a drifter/subnet has migrated to the new EE, the migrant drifter or the drifter that received the migrant subnet decides what values were missed while in transfer, based on timestamps and the assumption of relatively uniform time between EEs. The values not missed are then thrown away, the values missed are incorporated into the stream of datons generated by the appropriate input vertices.

Drifter execution environments Execution environment functionality is summarised in Table 2. These life-cycle operations directly correspond to those for drifters. For instance, upon receiving a drifter, its **on-arrival** handler is executed before passing control to the its **run** handler. We assume that drifter code is implemented in an identical library available at every EE.

The EE API provides support for generic inter-drifter and drifter-consumer messaging. Only asynchronous, non-blocking messaging is modelled, as it is most suitable to the unpredictable network environment. There are also operations that initiate/stop data source polling on behalf of drifters. Apart from the ability to cache polled values as described above, delegating polling to the EEs allows the packaging of multiple, simultaneous requests to the same data source from the same EE into a single protocol unit, thus minimising overhead.

Operations in support of data source discovery (needed to adapt to the population dynamics as described in Section 2) and network sensing infrastructure are not included. Both of these functions are likely to be co-located with EEs, and are the subject of ongoing work.

Communication protocols The commonly used communication protocol to retrieve the values of monitored variables in service management is SNMP [4], which is UDP-based. We model only one operation of the protocol, **get**, and two protocol data units (PDUs), GET and GET-RESPONSE. Multiple values can be requested with a single GET PDU and received in a single GET-RESPONSE PDU.

There is no single standard protocol used to transfer mobile agents between EEs and messages between mobile agents. Most of the proposed standards discuss a reliable, TCP-based protocol. Consequently, we adopt the Agent Transfer Protocol (ATP) [8] of the (open source) Aglets mobile agent platform, and extend it as needed. It is modelled after HTTP and is TCP-based, and forms the basis for our simulated Drifter Transfer Protocol (DTP).

ATP defines four standard request methods for agent services: **dispatch**, **retract**, **fetch**, and **message**. DTP models only the **dispatch** and **message** methods, message formats mirroring those of ATP. Similar to ATP, DTP is an EE-to-EE protocol (drifters communicate via host EEs), but consumers also use it to communicate with drifters (via EEs).

4. Emulating aggregation nets in *ns*

Drifter decision-making in aggregation nets (this part of the project is under way and is not reported in this paper) relies of a network sensing infrastructure to provide it with views of current and predictions of future end-to-end network characteristics. It also relies on current and predicted views of load loads on the computational resources. In the Grid architectures, this functionality is provided by the Grid middleware.

As having a sufficiently large, fully controlled, and heterogenous physical network to evaluate aggregation nets

Drifter life-cycle	Drifter-data source messaging	Inter-drifter messaging	Drifter-consumer messaging
create/dispose dispatch/receive	start-polling stop-polling	send-message-to-drifter receive-message-from-drifter	send-message-to-consumer receive-message-from-consumer

Table 2. Basic EE API

over is practically impossible, we turn to simulation. It is clear that apart from the aggregation nets model described above, we need to model the Grid-type services at some level. One approach could be using analytical models of the views that might be provided by such services, but it is safe to say that such models in the case of network resource utilisation are very preliminary at best. An alternative is to simulate the physical network as accurately as possible, and then use existing *Fabric* layer algorithms designed for real networks (such as those of the Network Weather Service) on the simulated network, whenever possible. The latter approach, in our view, will provide a greater degree of realism for our simulation.

The network simulator *ns* provides provides realistic, well-tested, and continuously improved models of different network technologies, protocols, and routing. It can simulate network dynamics and import real network topologies and traffic traces in the formats commonly used in networking research. Network sensing algorithms can operate on top of *ns* in a manner indistinguishable from the real network. *ns* does not provide facilities for simulating for simulating resource dynamics within individual hosts, but can be extended in this way, using an existing analytical model of host load.

ns is an event-driven simulator with two kinds of events, *packet*-events which correspond to the arrival of packets at physical network nodes, and *at*-events which are directly scheduled, and can be defined during simulation and based on the events in the simulation. This allows modelling of adaptive behaviour, and is one of the key advantages.

ns is extensible as simulation scenarios are arbitrary programs in a version of MIT Object Tcl (OTcl). The simulator is essentially a custom built OTcl interpreter based on the split-language model whereby objects can be implemented fully in C++ or OTcl, or partially in both. Implementing objects in OTcl does not require recompilation, but can affect performance of large simulations. Generally, low-level protocols are implemented in C++ and higher-level entities in OTcl.

ns includes support for passing application-specific data in simulated network packets, required by both simulated SNMP and DTP. In the simulated environment, there is no need to actually serialise/de-serialise drifters for transfer or transfer actual messages as every object is accessible. Rather, one only needs a correct estimate of the size of the serialised drifter or message to simulate transfer of an equal number of bytes. This number is then given to the network layer, and the packets of appropriate size are sent. The content of these packets does not have to conform to the application protocols simulated.

The mechanism that makes possible execution of drifter life-cycle events, message handling, or processing results of SNMP polls is based on handling message arrival and works as follows. A DTP **dispatch** message from an EE to an EE in fact contains a full OTcl command to execute the **receive-drifter** method on the target EE object, with the appropriate parameters. The DTP message handler in the EE class simply executes this command on message arrival. Scheduling and controlling operator net firing/backfiring is based on *at*-events, and involves manipulation of event lists by the drifter code. Operator nets entities and drifters are implemented as objects entirely in OTcl. Data sources, EEs and consumers are split objects, with message-handling code implemented in C++. SNMP is implemented on top of the UDP code, and DTP on top FullTCP code in *ns*.

All simulation scenarios for a drifter architecture/parameters under study are created with this general template:

- choose simulation time-frame,
- define/generate/import physical network topology and background traffic pattern/trace for chosen time-frame,
- define/generate a synthetic application,
- create data sources, EEs, and consumers and place them on physical nodes,
- partition the application based on partial knowledge of the number/parameters of EEs and network topology,
- schedule population dynamics events,
- choose EE load variation pattern,
- schedule start of background traffic, and
- schedule initial drifter creation.

The network topology and background traffic data can be defined manually for small networks, imported for actual WAN monitoring traces, or generated algorithmically. The topology includes loss models for each physical link, and the numbers and distributions of EEs/data sources/consumers can be specified by hand or selected at random. Initial partitioning of the application is influenced by the number of EEs known at application dispatch time, and optionally by the loads on the known EEs/network links. Population dynamics events include appearance/disappearance of data sources, and appear-

ance of new EEs during application run-time, and relocation/appearance/disappearance of consumers. Currently, we avoid EE fault tolerance issues and assume that EEs do not fail.

5. Conclusion and future work

As we have described, aggregation nets are a typical Grid application, whose *design* both becomes simpler with the assumption of Grid services, and can influence the *requirements* for such services. In this paper, we have provided a detailed model of aggregation nets and outlined the corresponding implementation in *ns*, which forms the base of our testbed.

Independent parts of the testbed not described in the paper include a *generator* of realistic synthetic monitoring applications, an automated *initial partitioner* for generated applications, and an emulation of some form of network sensing infrastructure in *ns*; all are under active development. In the course of construction of the testbed, we have found that although *ns* was not originally intended for application simulation, it can be extended to support such simulations, and is particularly helpful for network-aware applications that need to break network layer abstractions.

Future work will include selection of mostly reactive drifter architectures, mechanisms for inter-drifter coordination, automated generation of simulation scenarios, and simulation based on synthetic and real network topologies and traffic patterns/traces. We intend to apply our approach to a wider class of signal processing applications with naturally distributed sources of input data.

References

[1] E. Ashcroft. Dataflow and Eduction: Data-driven and Demand-driven Distributed Computation. In *Current Trends in Concurrency, LNCS 224*, pages 1–50. Springer-Verlag, 1986.

[2] E. Ashcroft, A. Faustini, and R. Jagannathan. *Multidimensional Programming*. Oxford University Press, New York and Oxford, 1995.

[3] S. Bajaj et al. Improving simulation for network research. Technical Report TR 99-702b, Computer Science Department, University of Southern California, September 1999.

[4] J. Case, R. Mindy, D. Partain, and B. Stewart. Introduction to Version 3 of the Internet-standard Network Management Framework. IETF, April 1999. RFC 2570.

[5] S. Frølund, M. Jain, and J. Pruyne. SoLOMon: Monitoring end-user service levels. In *Proceedings of the Sixth IFIP/IEEE International Symposium on Integrated Network Management*, Boston, Massachusetts, 1999.

[6] G. Goldszmidt and Y. Yemini. Distributed management by delegation. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, Vancouver, British Columbia, May 1995. IEEE Computer Society.

[7] R. Jagannathan. A descriptive and prescriptive model for dataflow semantics. Technical Report SRI-CSL-88-5, Computer Sciences Laboratory, SRI International, Menlo Park, California, 1988.

[8] D. Lange and M. Oshima. *Agent Transfer Protocol, ATP/0.1*. IBM Research Labs, Tokyo, http://www.trl.ibm.co.jp/aglets/atp/atp.htm, 1997.

[9] D. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison Wesley Longman, Reading, Massachusetts, 1998.

[10] R. Pinheiro, A. Poylisher, and H. Caldwell. Mobile agents for aggregation of network management data. In *Proceedings of ASA/MA'99*, pages 130–140. IEEE Computer Society, 1999.

[11] M. Ranganathan, A. Acharya, and J. Saltz. Distributed resource monitors for mobile objects. In *Proceedings of the Fifth International Workshop on Object Orientation in Operating Systems*, pages 19–23, Seattle, Washington, 1996.

[12] K. Schloegel, G. Karypis, and V. Kumar. Graph partitioning for high performance scientific simulations. In J. Dongarra, I. Foster, G. Fox, K. Kennedy, and A. White, editors, *CRPC Parallel Computing Handbook*. Morgan Kaufmann, New York, 2000.

[13] R. Wolski. Dynamically Forecasting Network Performance Using the Network Weather Service. Technical Report TR-CS96-494, Computer Science and Engineering Department, University of California, San Diego, 1998.