

DAUTI: Automated Universal Traffic Introspector

Jonathan Wellons^{*}
Department of Computer Science
Vanderbilt University
Nashville, Tennessee
wellons@gmail.com

John Wisneski
Department of Computer Science
Vanderbilt University
Nashville, Tennessee
john.p.wisneski@vanderbilt.edu

ABSTRACT

This paper introduces DAUTI, a network traffic simulator, applies it to a simulation research problem, and demonstrates the ease-of-use and applicability of functional languages to network simulation. DAUTI is an Open-Source, fully configurable, rapidly advancing, and efficient Scheme program that can be used to simulate Cluster load and job delay on any size of network, and provide basic statistics and analysis. In this paper, we use DAUTI to compute the relative performance of load distribution algorithms on a two-stage pipeline with heterogeneous cluster sizes. Specific applications include web servers, and industrial, research and financial batch processing. The choice of the Scheme language (a relative of Lisp) presented numerous opportunities to quickly produce compact and efficient code. This language selection incurred no obstacles.

General Terms

Functional Programming, Network Simulation and Analysis

Keywords

Scheme, Lisp, Functional Languages, Network Simulation, Network Analysis, Network Flow

1. INTRODUCTION

As networks grow to become the dominant means of service delivery, it becomes increasingly important that their full range of behavior is understood, dynamically and in advance of changing and building the network. As such, a set of powerful modeling tools is required, to enable the network engineer to simulate, plan and scale for worst-case traffic load.

Software productivity has always been strongly correlated with abstraction, a goal well served by Scheme, a language with abstract types and list manipulation, macros, functions

^{*}To whom correspondence should be addressed.

as first class data types and objects. DAUTI represents a case study in the true effectiveness of these features, and we seek to examine how each of them benefits a concrete tool in a domain typically occupied by low-level languages, such as C++.

This paper presents a tool we have just released: DAUTI (*DAUTI - Automated Universal Traffic Introspector* is a recursive acronym), which provides much of this needed functionality. Although much remains to be done (see Further Work), we have used DAUTI to verify and measure the differences between a number of well-understood load distribution algorithms in a pipelined, staged network that closely matches many physical networks (such as some HTTP services). An introduction to networks of this kind and empirical models for analyzing topologies may be found in [3].

These data help confirm that DAUTI operates within the realm of plausibility and has immediate applications on networks meeting its conditions for job arrival and execution time, pdf (Probability Distribution Function), and network model. Furthermore, DAUTI has been extensively tested in development and released as an Open Source application.

In section 2, we provide theoretical background to justify our choice of experiments and implementation decisions. Section 3 places each of our default algorithms in this context. Section 4 is a technical look at the mechanics and usage of DAUTI, section 5 gives our simulation results, section 6 lists some tantalizing areas for future work and section 7 offers a conclusion.

2. MOTIVATION

The value provided by DAUTI is in modeling networks in several situations.

- Prior to Building the Network

Many Server farms are designed based on guesstimates and hastily deployed to capitalize on market opportunities. It is hoped that the ease-of-use and flexibility of DAUTI make it a cost-effective step in this process of designing a network, particularly once it has built-in accurate models of most common computer equipment.

- Growing a Network

It is a common industrial practice to grow networks organically, often in “safe” incremental steps that ignore

the benefits of comprehensive reorganization. When a network begins to approach capacity, it is essential to understand which network layers are bottlenecks, tune the model to ensure it properly matches the physical network, and then cheaply and quickly explore possible reorganizations of large and small scales.

- Troubleshooting

DAUTI has an application in diagnosing network failures. In response to a sudden rise in rejected jobs, the software may be used to trace paths of high traffic and identify hot spots in the graph topology, even when they are not exact location of the rejection. This provides another dimension on top of that offered by standard tracing tools.

3. ALGORITHMIC CONTEXT

Singhal and Shivaratri [2] give three classes of algorithms for distributing load between computers. *Static algorithms* distribute load based on hard-coded information about the system. *Dynamic algorithms* use information about the current state of the system to determine which computers will receive jobs. *Adaptive algorithms* are a subclass of dynamic algorithms that can change how the algorithm works based on current conditions. The example of an adaptive algorithm given in Singhal and Shivaratri is an algorithm that ceases collecting state information when the system load is too high [2].

Singhal and Shivaratri also discuss the difference between preemptive and nonpreemptive transfers. A preemptive transfer suspends a process in order to transfer it with its current state to a new computer. A nonpreemptive transfer is a transfer that occurs before a process starts executing. Preemptive transfers are more expensive than nonpreemptive transfers because of the additional state information that must be sent in a preemptive transfer [2].

In this taxonomy, a load distributing algorithm has four parts: the transfer policy, the selection policy, the location policy, and the information policy. A node's transfer policy determines whether that node should be a sender, a receiver, or neither one. A node's selection policy is its mechanism for determining which job to transfer to another computer. A location policy is used by a node to determine which other nodes are available as senders if the node is a receiver, and which nodes are available as receivers if the node is a sender. The location policy also determines to which of the eligible nodes a job will be transferred [2].

The information policy of a node determines when the node sends or collects state information. An information policy can be demand driven, periodic, or state change driven. A node using a demand-driven information policy will collect information about the states of other nodes when it needs that information at the time it becomes a sender or receiver. A demand-driven policy may be set up to collect information when a node becomes a sender, when a node becomes a receiver, or when a node becomes either a sender or a receiver. In a periodic information policy, nodes send and receive state information at regular intervals. In a state-change-driven information policy a node sends its new state, whether it is a

sender, receiver, or neither, to a central server in a centralized policy or to its peers in a decentralized policy [2].

4. SELECTED ALGORITHMS

Four algorithms were used to determine to which successor node a job should be sent: random successor, random successor with space in queue, successor with shortest queue, and successor with longest nonfull queue. For all algorithms, when a job is sent to a node with a full queue, the job must be return to the originating application so that it may be reattempted at a later date. This reattempt may be by hand, in the case of web browsing, or it may be automated in the case of industrial batch processing. We will deem this a *job rejection* or *delay* and the job is removed from the network. Any job ready to be sent when the queues of all successors were full is rejected.

All four algorithms have the same transfer policy. All nodes become senders once they have finished processing their current job. All nodes are always receivers. Similarly, all four algorithms have the same selection policy. Jobs are transferred immediately once finished.

Each algorithm has its own location policy. The random successor algorithm choses a node at random from its complete list of successor nodes. The random successor with non-full queue algorithm filters the list to avoid sending the job to a node with a full queue and randomly selects from among the remaining nodes. We dub this distinction *intelligence*. The successor with shortest queue algorithm transfers the job to the successor with the lowest ratio of jobs in queue to queue capacity. The successor with longest queue transfers the job to the successor with the highest ratio of jobs in queue divided by queue capacity. Both the successor with shortest queue and successor with longest queue algorithm break ties by transferring the job to the tied node closest to the beginning of the sender's successor list. The tie-breaking method can be important in some graphs although not in the network studied in this paper. It is quite simple for a user to change tie-breaking to random.

Three of the algorithms, random successor with space in queue, successor with shortest queue, and successor with longest queue, share an information policy. They use information from a global state space to make their decisions, making them a combination of demand-driven, and state-change-driven information policies. The state is stored in a central location as in a centralized state-change driven information policy, but it is not retrieved from that central location until a node becomes a sender, as in a sender-initiated demand-driven information policy. Since the successor functions do not need access to the entire system state, a physical network modeled by DAUTI need only transfer a much smaller amount of data. The random successor algorithm has no information policy because it never obtains information about the state of the system once the simulation begins.

5. IMPLEMENTATION

The initial version of DAUTI consists of only 388 lines of Scheme, of which 67 are libraries, 30 dedicated to the command-line user interface and the rest to the engine, configurations, and statistical facilities. Using a special input format, traffic-

flow networks of arbitrary size and complexity may be simulated, including cyclic graphs.

The choice of Scheme provides several principle advantages:

- Recursive Data Structures

It is mandatory that any network flow modeling software allow jobs to potentially revisit nodes that have already processed them, for instance, in Industrial Workflow models, Multi-Pass Parallelized Compilation and many species of network applications. Yet, many languages, particularly on a lower level most associated with network modeling make specifying such graphs a non-trivial task. This implementation makes cyclic graphs immediately representable.

- No Redundant Code

The flexibility of Scheme stemming from treating code as data permits extensive reuse, of the core engine functions, the statistical functions, successor algorithms and even the output formatter. An informal visual inspection of the code confirms that essentially nothing is duplicated and that the code is fully normalized.

- Compact Code

Since many operations in the running of the program require identifying nodes by property (number of jobs queued, relative speed, number of connections), Scheme's `map` and list foundations allow us treat the nodes with a *set abstraction*. The same principle dramatically simplifies computation of the final statistical tallies.

- Graph Templates that are `lambda` Functions

In order to sequentially test many complex topologies and successor-selection algorithms that differ only in details, metatemplates are defined as functions that take successor functions as arguments and “plug them in” to the appropriate locations on all nodes in the model.

5.1 Usage

Each node n is specified by the user with 6 parameters: its ID, successors, tags (e.g if n is a starting or stopping state for jobs), buffer capacity, speed, and a user-definable successor function that describes how n is to decide which of its successors will receive a job when n has finished its stage. Note that attempting to place a job in a node with a full buffer constitutes a *job rejection*. This is the only way jobs can be removed from the system without finishing. Observe also that jobs can be rejected as soon as they arrive, when the assigned start node already has a full buffer. One of the most powerful features of DAUTI is flexibility of successor functions. Each of the algorithms described in section 3 are built-in: shortest-queue, random-eligible, longest-queue (to establish a upper case bound on an intelligent algorithm), and naive-random, (allocating without regard to full buffers which can cause job rejections even when idle successors exist). User-defined successor functions have access to the full system state, and as such, are subjected to no unnatural or contrived limits.

5.2 Engine

DAUTI operates by managing an internal list of all job counts at every node and assuming that jobs finish and arrive with (possibly different) exponential distributions. This distribution is chosen because it models many natural heterogeneous distributions and has extremely favorable statistical properties. Where there is a need to model pdfs wildly different from the exponential distribution, DAUTI is not recommended. Due to the internal mechanics of DAUTI, this change would not be easy to make.

The key aspect of the exponential pdf is *memorylessness*. Observe that in a normal state where there are n start nodes and an inward flow rate of F at each, the aggregate inward flow rate is jF . Further, when there are i jobs-holding nodes their total execution rate is $\sum n_{speed}$, where n ranges over all nodes with jobs. Note that these sets may overlap and constitute i possible new job events and n possible job finish events. Each of these $i + j$ incidents has a probability of occurring first of its own rate divided by the sum of the rates in the whole system. Therefore, the next event to occur can be drawn randomly with the correct bias, as we know that the time-until-next-event pdf is itself exponential, allowing us to calculate the expected time for this to occur. DAUTI therefore operates *discretely*, by events, rather than continuously, thus avoiding the fluid model of network flow, and its pitfalls. A discussion of tradeoffs between the Fluid model and the discrete model is outside the scope of this article; please see [1].

The DAUTI function that selects the next event in the system is given in Figure 1.

As DAUTI runs forward in simulated time, it tracks the number of events, the number of finished, unresolved (ie., still in the system), and rejected jobs, and the estimated elapsed time for each trial until a given number of job transitions has occurred. Any number of trials may be run at once, with only aggregate statistics returned and source code is easily configured to allow some variables to range over the trials, for instance running 1000 trials each of 1000 transitions for all inward flow rates between 1 and 60, with random successor selection. DAUTI's command line interface makes such a call easy to write and observe. Under the current interface, 1 - 60 could be set as the inward flow range and such a command could be given as follows:

```
> mzscheme -f dauti.ss 1000 100 r
```

Networks are specified using a flexible and programmable data structure, based on `node structs`, which is extensible in Scheme itself. The internal representation of nodes is quite simple.

The successor-function can be any function, user-defined or built-in, that when given a list of possible successor nodes, a node, and the current state, choose a node to send the next job to. For example, a successor function that always alternates successors can be given as shown in Figure 3. The more sophisticated shortest-queue-successor function is shown in Figure 4.

For our experiments below, we simply created a template

```
(define next-event
  (lambda ()
    (letrec ((current-queues (with-jobs)
              (queues-sum (apply + (map node-speed current-queues)))
              (pool-size (+ (* (length start-states) inflow) queues-sum))
              (quantum (exact->inexact (/ 1 pool-size)))
              (chance (random pool-size))
              (type (if (>= chance queues-sum) 'new 'finish))
              (selected-node (if (>= chance queues-sum) (random-select start-states)
                                (node-id (list-index (summed-index chance (map node-speed current-queues) 0)
                                                    current-queues))))))
      (list type selected-node quantum))))
```

Figure 1: Randomly Choosing the Next Event

```
(define shortest-queue-successor
  (lambda (next-positions next current-state)
    (letrec ((with-capacities
              (map (lambda (pos)
                    (letrec ((c (node-capacity (id->node pos)))
                            (current-size (queue-size current-state pos))
                            (fraction-filled (exact->inexact (/ current-size c))))
                      (list pos fraction-filled)))
                  next-positions))
      (result (inexact->exact (car (min-f with-capacities cadr (car with-capacities))))))
    result)))
```

Figure 4: Shortest-Queue successor function

```
(define-struct node
  (id
   successors
   fields ; a list of extra parameters
   capacity
   speed
   successor-function
  ))
```

Figure 2: Use of Scheme's structs

```
(define alternating-successor-generator
  (lambda ()
    (letrec ((pos 0)
              (lambda (next-ps node state)
                (let ((this-p (modulo pos
                                      (length next-ps))))
                  (set! pos (+ pos 1))
                  (vector-ref (list->vector next-ps)
                              this-p))))))
```

Figure 3: Alternating successor function

function for our network that could be passed parameters indicating which successor function to use. As such, no copy and paste was needed. This template is packaged with the source as an example.

For example, the model later used in this paper could be encoded as shown in Figure 5 below. The topology is show graphically in Figure 6.

```
(define inflow 60)
(define model
  (make-node 0 (seq 2 7) (list 'start) 3 30 sf1)
  (make-node 1 (seq 2 7) (list 'start) 3 30 sf2)
  (make-node 2 (list ) (list 'finish) 2 10 \#f)
  (make-node 3 (list ) (list 'finish) 2 10 \#f)
  (make-node 4 (list ) (list 'finish) 2 10 \#f)
  (make-node 5 (list ) (list 'finish) 2 10 \#f)
  (make-node 6 (list ) (list 'finish) 2 10 \#f)
  (make-node 7 (list ) (list 'finish) 2 10 \#f)
  ))
```

Figure 5: Sample Network Configuration

It is expected that a non-novice user will compress this with the Lisp family's uniquely powerful macros. A set of builtin macros are likely to appear in the next version of DAUTI. Given that DAUTI places no limits on the size of graphs and that networks with thousands of nodes commonly need to be modeled, this is particularly useful to simplify the configuration and editing process. In the case of the Figure 5, it is easiest to compress the lines corresponding to nodes

0 through 1 to one line and nodes 2 through 7 also to one line, to save editor space and allow the groups to be updated at once.

Consistent with research tradition in the field of Lisp programming, DAUTI is fully open source and freely available. DAUTI's development status is very active with rapid resolution of bugs and feature requests. However, the number of features that have been requested for future versions is growing very rapidly, and it remains an interesting experiment to view how the flexibility and power of functional, macro-backed programming will meet scaling in this case.

5.3 Design Choices

DAUTI supports buffer capacities for individual nodes rather than the system as a whole or subsets of nodes, as the latter cases were deemed unrealistic. Similarly, DAUTI does not support shared queues, since the authors have not seen these widespread in practice. All the jobs within the system are considered to have an identical exponential distribution, since many traffic systems serve a wide range of requests, with effectively unbounded, but rapidly-decaying, long-tailed size distribution.

5.4 Staging Pools

Although DAUTI can be used with any network topology, we have found it particularly enlightening when examining *Staged Traffic*, where jobs must visit each of a family of services, in order. This closely models various real-world applications, such as Application stacks on a server (e.g., Apache to MySQL) or web services. In a typical Staged Traffic scenario, each job is represented by a dedicated pool of servers, connected such that each server in a pool P_i , may send jobs to all servers in P_{i+1} and receive jobs from all servers in P_{i-1} . We assume in staged simulations that incoming jobs are distributed randomly among starting servers to mirror common client/server applications. See Figure 6, for the example used in our simulations.

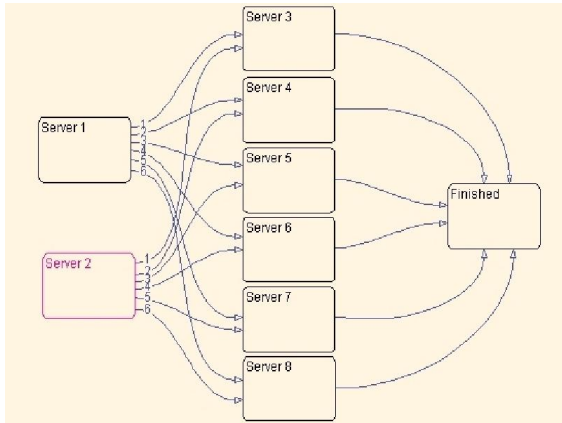


Figure 6: 2 Layer Server Pool

6. RESULTS

Using DAUTI, we simulated the topology given above (with parameters of 1000 trials and 1000 transitions) for 120 values of inward bound flow. The magnitude of flow through the diagram is inconsequential, only the relative weights matter

so no units are used in this paper. To control for load as the independent variable, we fix the server speeds inside the network, giving the two front-end servers rates of 30 each (buffer size of 3), and each of the backend servers buffer capacities of 10 (buffer size of 2), for an aggregate throughput of 60 per stage. The incoming load is allowed to range from 1 to 120, to reflect a range of underloading to severe overload. Observe that rejected jobs can occur with any non-zero inward flow, due to the random nature of the arriving jobs. Each flow rate was tested using the 4 default algorithms described above: Naive Random, Random Eligible, Shortest Queue and Longest Queue. Performance is measured by the fraction of incoming jobs that are rejected.

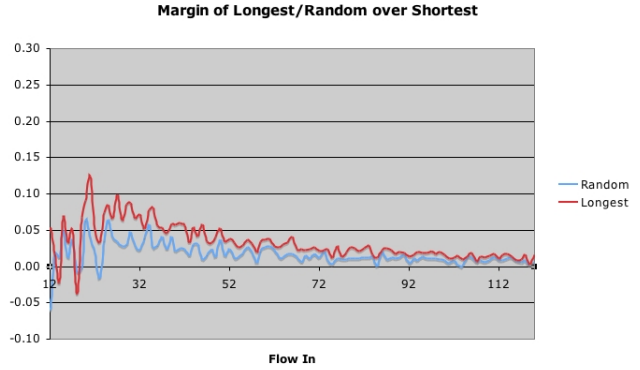


Figure 7: Relative Algorithm Performance

As expected (Figure 8), for very low load, all algorithms perform equally well. In this situation, there is normally no contention for a CPU and the potential recipient buffers are normally all empty. As a result, the algorithms have few opportunities to distinguish themselves and few jobs are rejected. Note that job rejections remain rare even when throughput begins to suffer from minimally full queues, because jobs will not be rejected until an entire queue is full. If latency were being measured, we would see degradation earlier (see Further Work). The maximum discrepancy between Longest Queue and Shortest occurs at inward flow of 27 and has a relative magnitude of approximately 10%. This can be most clearly seen if we show Longest Queue and Random as a marginal fraction of Shortest Queue, as in Figure 7.

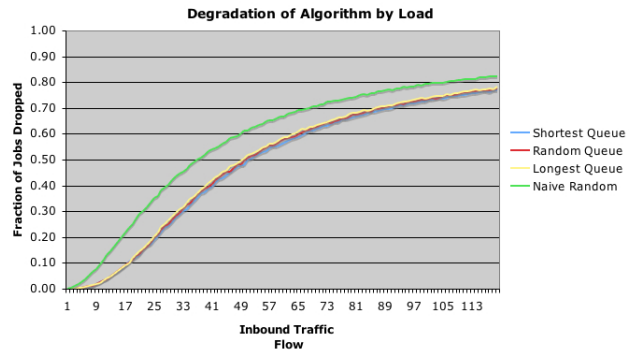


Figure 8: Degradation

An entirely different perspective may be seen in Figure 9. Here, rather than taking the average, we have expanded out the full range of lost jobs for each Algorithm for one value of inward flow. As expected the curves for Longest Queue are shifted to the right of Shortest Queue, and Random Queue is roughly in the middle.

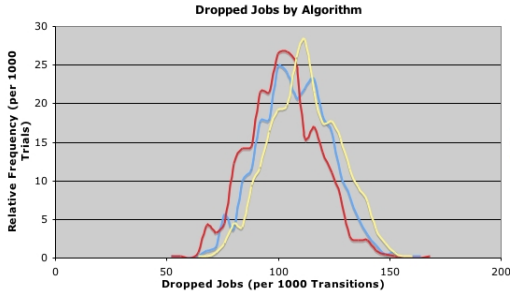


Figure 9: Relative Algorithm Performance

7. FUTURE WORK

- Direct Comparison to Other Languages

A preliminary inspection has shown that network tools written in C++ are far more verbose and difficult to write and maintain for equivalent functionality. It would be desirable to formalize this observation, but side-by-side code comparison for emergent or intangible features is notoriously hard. Nevertheless, experts in other languages may be asked to clone DAUTI to produce rough observations of time spent in development, ease of use, and maintainability.

- Depth of Evaluation for Successor Functions

It would be instructive and helpful to add a default successor algorithm that examined load further *downstream* than simply one level. Since the successor functions have access to the entire data model, the API supports this as a user-definable feature. Nevertheless, we believe it would be universally useful enough, and interesting from a comparative perspective to justify including in the default distribution.

- GUI support

DAUTI is written in Scheme, a sophisticated high-level language (it is a testament to Scheme’s sophistication that DAUTI requires so few lines) with GUI support. Improving the interface would improve user-friendliness and increase the reach and utility.

- Shifting Pools

It can often be difficult to predict in advance how much load each stage will undergo, and as a result, many modern architectures allow nodes to dynamically change their service type. It would be worthwhile to incorporate this into DAUTI, perhaps by allowing the successor list the option to be a function that returns a list in addition to being the lists itself.

- Speed Optimizations

The authors found DAUTI sufficiently rapid for up to 1000 trials of 1000 transitions with the given network

(8 nodes) illustrated in the results section. However, in light of human curiosity and practical reasons to examine extremely large networks over many tunable parameters, more improvements should be made. All simulations were run on 2.5-year-old 1.5 GHz Macintosh Powerbook. On this hardware, the simulations above (per inward flow value) finished in approximately 69 seconds of user time.

- Success Metrics

DAUTI currently uses job rejections as a (negative) success metric because its high applicability to many batch and client/server applications. Nevertheless, many other applications rely heavily on time delays and it would be straightforward to add this to DAUTI’s output statistics due to the self-contained nature of Scheme functions.

- Objective “Hallway Usability Testing”

All attempts by the authors to modify and extend DAUTI have succeeded with very few required deltas to the code. In order to more accurately gauge the ease of configuring and altering DAUTI by scientists that did not write the program, an experiment with objective subjects exploring use cases of their choosing is currently underway.

8. CONCLUSION

DAUTI, a versatile and growing network simulation tool, shows promise in modeling specific kinds of networks and understanding the expected load variations and extremes. In this paper, it has been used to experimentally verify and measure the superiority of Shortest Queue, known intuitively to be better. For the network in question, the measured variability for intelligent algorithms peaks at 10%, a significant, but not overwhelming value.

The authors view DAUTI as a case study in the practicality of a Lisp-like language for discrete simulation. Scheme turned out to provide an extremely solid basis in clean, efficient, Object-Oriented software engineering, allowing the first version of the entire package to be produced in a handful of programmer days.

9. REFERENCES

- [1] Dongyu Qiu and R. Srikant. Modeling and performance analysis of bittorrent-like peer-to-peer networks. In *SIGCOMM*, 2004.
- [2] Niranjana G. Shivaratri and Mukesh Singhal. *Advanced Concepts in Operating Systems: Distributed, Database, and Multiprocessor Operating Systems*, chapter 11: Distributed Scheduling, pages 263–266. McGraw-Hill College, 1994.
- [3] Christopher Stewart and Kai Shen. Performance modeling and system management for multi-component online services. In *NSDI*, 2005.