Implementing a Genetic Algorithm for Cognitive Packet Networking

Jeremy Lainé

Ecole polytechnique
Palaiseau, France

4th July 2002

Contents

List of Figures

Introduction

Routing of packets in networks requires that a path be selected either dynamically while the packets are being forwarded, or statically (in advance) as in source routing from a source node to a destination. Typically, routes are selected so that some criterion of performance can be satisfied, in addition to the most elementary need of conveying data from a specified source to a specified destination. Recently, the University of Central Florida has developed a Quality of Service (QoS) driven routing protocol called "Cognitive Packet Network" (CPN) which dynamically selects paths through a store and forward packet network so as to provide best effort QoS to route peer-to-peer user traffic. This technique uses smart packets to select routes based on the user's QoS requirements. In this paper CPN's path discovery process is extended to include a genetic algorithm which can help discover new paths that may not have been discovered by smart packets. We propose a genetic algorithm approach to the construction and selection of routes and apply it to the case where the desired QoS is as short a delay as possible. We detail the implementation of the algorithm in the Cognitive Packet Network test-bed and report the resulting QoS measurements.

1 Constraints and implementation choices

1.1 Goal and principle of the GA

In its current implementation, the Cognitive Packet Network (CPN) relies a set of lightweight packets, 'Smart Packets' (SPs), to explore new network routes and bring back information on these explored routes pertaining to the current Quality of Service (QoS) constraints. A wealth of information is brought back in this manner but it is only partly exploited by the CPN algorithm. The objective of the genetic algorithm is to assist the CPN in its routing decisions by doing some extra processing on this information. The way the GA works is by combining existing routes which share a common node and destination in order to generate new routes and selecting the routes which are more efficient in terms of achieving the QoS goal. This processing is done only at the source.

1.2 Constraints

1.2.1 System load and delays

One must bear in mind that *the GA processing is not critical for CPN to function*. Should very high system loads be encountered, the CPN module must prevail even if the GA processing is put on standby. The GA must therefore run as a separate process, be optional, disconnectable at will and must not inter-operate with the CPN algorithm in a synchronous fashion. As we are trying to improve the performance of the network, the presence of the GA should obviously not introduce extra delays.

1.2.2 Data sharing

The GA algorithm operates by doing a "crossovers" between pairs of paths and periodically discarding the paths which have the lowest fitness, for example the highest delay. In order for this selection to reflect the current state of the network, it can be interesting to base these calculations on the latest data available. For example let us consider two paths A and B which share some common hops.

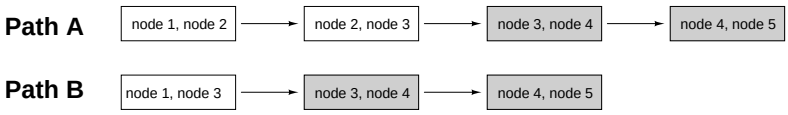

Figure 1: paths sharing common hops

When we evaluate the fitness of A and B in order to compare them, for this comparison to have a meaning, the underlying measurements on the hops (delay, loss) must reflect *the same network conditions*. As these paths were very likely brought back at different times, it should be possible to update the measurements for hops as new information is brought back from the network. As duplicating data is not desirable, the GA should have a pool of available hop measurements and paths should just be a collection of references to hops in this pool.

1.2.3 Development environment

Another important constraint on the GA implementation is that, like the CPN module, it should be portable to embedded systems with little or no changes. With this in mind, the GA was implemented in ANSI C and use of standard GNU tools insure smooth cross-platform compatibility.

Finally, the development of the Genetic Algorithm has to be repositioned in its full context: it is an extension of an existing and ongoing project. This means that the code is constantly being improved and proper versioning of the source code is an imperative. During the course of the development of the GA, the CPN project underwent a major reorganisation which resulted in the setup a fully-fledged concurrent-development environment.

1.3 Key implementation choices

1.3.1 The GA daemon

The size of the data structures alone that the GA needs to manipulate made a kernel-level implementation seem an unlikely choice. Also the CPN module already implements a data exchange system with user-land applications, which is for example used when configuring the network parameters or polling the module for statistics. Finally, as it has been stated above, the GA processing should be done in the background. For all these reasons, the GA algorithm has been implemented as a system daemon and will be referred to as "the GA daemon" from here on.

1.3.2 Data structure considerations

The relational nature of the data being processed made it was tempting to use a database and have the GA daemon run requests through the database engine. Both the latency factor and the end goal of CPN which is implementation on embedded systems ruled out this option. The data structures of the GA daemon nevertheless reflects the relations they share through the use of pointers, keeping both memory footprint and processing overhead low.

1.4 Behaviour of CPN module and GA daemon

1.4.1 Original behaviour of the CPN module

When the CPN module receives a request from a program to send data to a given destination, it starts by checking the *dumb packet route repository (DPRR)* for a known path to the destination. If none is available, it will fire a smart packet to try to discover a route to the destination. If on the other hand a known route exits, the module send a dumb packet along that path and with a certain probability fires an additional smart packet to bring back another route. As smart and dumb packets travel on the network, they collect timestamps from the nodes they go through.

When either a smart or a dumb packet reaches its destination, the route it used is stripped of any loops and an acknowledgement (ACK) packet is sent back to the source along the reverse route. As the ACK travels back toward the source, nodes calculate their round-trip delay to the destination by subtracting the timestamp they left on the original smart or dumb packet from the current timestamp and store this measurement data in the packet. When the ACK reaches the original source, the CPN modules updates the dumb packet route repository, replacing any existing route to the same destination and making the new route available for future dumb packets.

1.4.2 Behaviour of the GA-enabled module

The GA-enabled module works in a similar fashion to the regular CPN module, the main changes concern the handling of the ACK when it reaches the source. However, a modification to the way CPN generates and sends ACKs back was needed for GA to function properly. While CPN deals with entire paths, GA operates at a finer level and needs accurate measurements on individual hops

and this means that caution needs to be taken when stripping a route of loops. Loops are no longer removed at the destination but instead as the ACK travels back, the nodes on the path check if they were present twice or more in the original packet's path and if that is the case, they remove the loop and adjust the timestamps of nodes between itself and the source to compensate for the extra delay introduced by the loop.

When the ACK reaches the source, we have a complete path to the destination and the measurements for each hop. This data is put into a FIFO buffer for the GA daemon to process whenever it becomes available. Unlike the regular CPN module, the GA-enabled module does not update the dumb packet route repository unless there is no route for the current destination or the GA daemon is dead (defined as "if if the GA daemon has not talked to the module for the past second").

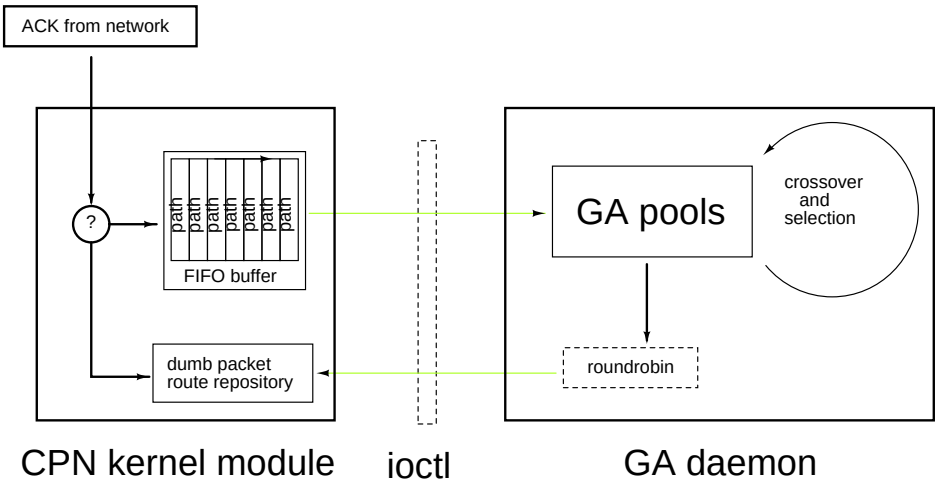

Figure 2: interaction between module and daemon

1.4.3 Behaviour of the daemon

The GA daemon is fired up at any time when the CPN module is loaded with no initial knowledge of the network. It consists of a main loop which is in charge of polling the kernel for new paths, checking its internal data structures for size and consistency, selecting individuals for crossover and doing the actual crossover and periodically updating the CPN module's dumb packet route repository.

2 Internals of the GA daemon

2.1 Data structures

2.1.1 General ideas

As has been previously stated, the GA daemon's function is to assist the CPN by providing it with "the big picture", that is not just the currently used path to a given destination but a set of alternative routes with their associated fitness.

In order to make maximum use of the measurement data, the information is broken down into its smallest constitutive element, the measurement for one hop. The data structure storing the pair of nodes which constitute the hop and the corresponding measurement information is referred to as a *gene* in the rest of this document. A path is a collection of hops or genes and is referred to as an *individual*. To make it easier to select individuals for crossover, the individuals are arranged in subpopulations based on their destination.

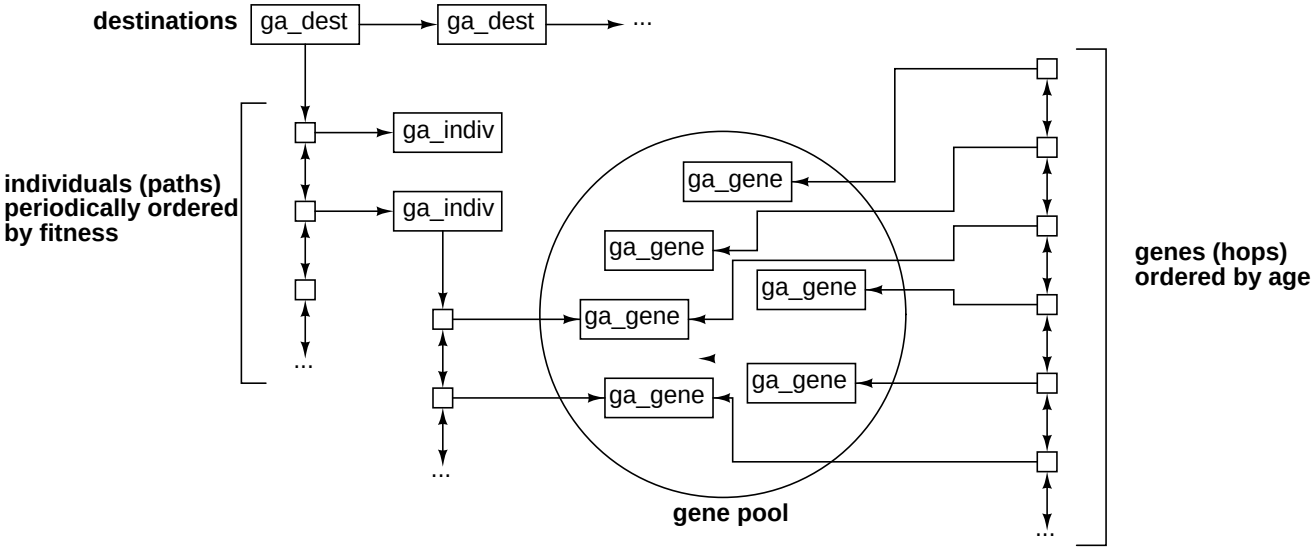

Figure 3: principal data structures of the GA daemon

2.1.2 Data exchange between GA daemon and CPN module

The data exchange operations between the CPN kernel module and the GA daemon is bidirectional as the module passes measurements to the daemon and in return the daemon periodically updates the module's route repository with the paths who have the best fitness at that time. The GA daemon is always the initiator of the data exchange so that the kernel module does not need to keep track of the presence or absence of the daemon to route CPN traffic. This way we also eliminate lockups which would occur if the kernel were to probe the daemon while it is sleeping.

In both exchanges, the data concerns paths so the data format is the same. For the sake of efficiency and modularity, the CPN module only has a minimum knowledge of the internals of the GA daemon and data is exchanged in the form of simplified individuals. As data exchange with the kernel module is done through a byte buffer, the exchange format consists of a header bearing timestamp, source node, number of nodes and a byte for flags followed by a succession of node/measurement pairs.

2.1.3 Hop measurements (*genes*)

Hop measurements are represented by a data structure which stores a timestamp, the two nodes between which the measurement took place and the actual delay measurement. As newer measurements are sent over by the CPN module, genes are updated in place to reflect the new network conditions, and their timestamp is updated as well. The timestamp allows us to periodically rid the gene pool of obsolete genes, so that the GA engine does not make decisions based on obsolete measurements.

These hop measurements or genes are referenced in two ways. The first is a linked list of pointers to all the genes ordered by increasing age, which puts the most recent ones on top available for crossover operations and the oldest ones on the bottom, ready for deletion if the gene pool grows too large. The second way genes are referenced is in the GA individuals, that is paths, who contain the considered gene. In order to make deleting obsolete genes and the associated individuals fast, the genes store a list of the individuals which use them.

2.1.4 From hops to paths (*individuals*)

The GA individuals each correspond to a route and their internal representation of this route is a linked list of pointers to genes. An individual does not store its fitness as the delays associated with the hops that built up the path will change over time. The individuals are arranged in subpopulations, one for each destination, and within these subpopulations they are periodically ordered by fitness. This is done just before the subpopulations are checked for size, so that when the size limit is reached, the less efficient paths are the ones that get dropped.

2.2 Managing the pools

As has been pointed out, the GA structures exhibit a high degree of cross-referencing so care was taken to handle the structures in a consistent and reliable fashion. Also, if the GA daemon is to run on a limited amount of memory, it must be able to control the size of its data structures.

2.2.1 Merging information into the pools

New paths appear make there way into the pools in two ways : either we get a path from the kernel after it was brought back by an ACK or we generate a new path by crossover. When a path is received from the kernel it is broken down into hops and we generate the equivalent GA individual. We go over the genes that build up the individual and we compare them to those existing in the gene pool. If we find a gene describing the same hop, we update the timestamp and delay measurement for this gene and move the gene to the top of the gene pool, then discard the temporary individual's gene and replace it by a reference to the pool's corresponding gene. Otherwise we add the gene to the top of the pool. By the time we reach the end of the genotype, all of the individual's genes are in the gene pool and we know if an identical individual exists. If it does we can discard the temporary individual, otherwise we add it to appropriate subpopulation based on its destination.

2.2.2 Controlling pool sizes

Control of the size of the gene and individual pools is achieved by periodically running a cleanup function which eliminates obsolete or excess genes and individuals. The cleanup is done in two steps, first 'marking' the items to be deleted then actually deallocating them, a gene being marked for deletion when no individuals are registered as a user of that gene.

The gene pool is processed first by a function which runs over the genes, counting them as it goes. If a gene is obsolete or if the maximum count has been reached, the individuals containing this gene are unlinked from all their constitutive genes. Next we order the subpopulation of individuals for each destination by evaluating the fitness of each individual at that time, dropping the individuals which were marked dead as we go. If we have too many individuals, we unlink the excess individuals from their genes and discard them. All that is left to do now is to discard all the genes which are marked for deletion, that is which have no more individuals registered as using them.

2.3 Crossover related operations

The following functions form the core of the path selection process which precedes the crossover operation. Their role is to return a pair of individuals which are suitable for crossover. As we will see, depending on the situation the GA daemon's main loop either requests a "match" for a specific individual or just asks for a pair of individuals to crossover.

2.3.1 Pre-crossover selection : ga_indiv_match(ga_indiv *indiv)

Given an individual which we shall call *indiv*, this function tries to find another individual suitable for crossover, that is an individual which has the same source, the same destination and a common node in the middle.

The search is done by going over *indiv*'s nodes from source to destination, excluding the source and the destination themselves. Let *node* be the node we are considering. We go over the gene pool, looking for a gene which contains *node*. Among the individuals containing this gene, if one has the same destination as *indiv*, we have found our pair of individuals and know what node they share, otherwise the search continues. As the gene pool is ordered by increasing age, we pay more attention to the hops for which we have recent measurements. This encourages processing of data while it is fresh and means that the fitness of the individual resulting from the crossover is likely to reflect the current network conditions.

2.3.2 Pre-crossover selection : ga_crossover_select()

While the ga_crossover_match function is useful for matching a given path, in over 90% of its cycles, the GA daemon is not looking for a specific match, we simply want a pair of paths which can be crossed over.

As we have several destinations to consider and as over time we want processing to be done for all of them, the ga_crossover_select function starts by picking a destination in a round-robin fashion. Within the subpopulation of individuals corresponding to that destination, we also pick an individual according to a round-robin discipline. We then call ga_crossover_match to find a suitable match for that individual and the node they have in common. If this fails to yield an individual we shift the round-robin.

2.3.3 Crossover : ga_crossover(ga_ipair *ipair)

Once we have selected the pair of individuals which will be used for the crossover and the node at which crossover occurs, the operation is simple. We create two new individuals and copy beginning of one of the parent paths up to the common node into one and the beginning of the other parent into the other then switch and continue. We evaluate the performance of the offspring, discard the worst one and merge the best one into the subpopulation for the current destination.

2.4 Main loop

2.4.1 Polling the *CPN* module for new paths

Upon receiving an ACK packet, the CPN module makes the received route available to the daemon but putting it into a FIFO buffer. This buffer has a fixed size and if the daemon does not request the data - for example in the case of a very high system load - new data pushes the old data out. There is a restriction to this behaviour due to the dual nature of the measurements : the ACKs generated by smart packets most likely bring back new paths while the ACKs generated by the dumb packets carry updated measurements on known paths. Diversity in the individual population is highly desirable both in order to have a set of alternative routes if the current route becomes saturated and in order to enhance the chances of crossover producing interesting new individuals.

For the same reason, the GA daemon treats paths brought back by smart packets in a different fashion. To make this possible, along with the path itself, the daemon receives a set of flags, the only currently used being the 'priority' flag which is set if the path was generated by a smart packet.

2.4.2 Crossover operations

After polling the kernel, the next step in the main loop is to proceed with crossover operations. If we received a path from the kernel in the current cycle and that path bears the priority flag, we call the ga_crossover_match function to try to cross the new path over with an existing path. If this does not yield a pair of individuals we call ga_crossover_select to try to continue round-robin processing of the individuals where we last left off. If either of these methods yielded a pair of individuals we call the crossover function.

2.4.3 Cleaning up the pools

The cleanup function is not run on every cycle of the GA daemon but when it called, this occurs just after the crossover operation. After the cleanup we have compensated for whatever excess growth the pools had known since the last cleanup. The paths for each destination have also been ordered by fitness so we are ready to send results back to the kernel.

2.4.4 Sending GA results back to the CPN module

In a first implementation, the GA daemon would always send the kernel the best path for a given destination, but after some testing this was changed. The two problems encountered were route saturation and a depletion of the individual pool.

The first problem is easy to understand, and it also occurs with regular CPN : if we do not change the route in the dumb packet repository for a given destination, all dumb packets - which are both the largest packets on the network since they are the information bearers and the most numerous - fired for this destination will follow the same route and in the case of heavy traffic they will saturate that route. In the case of GA this is counterbalanced by the fact that the dumb packets will generate ACKs with rising delays and the route will be abandoned once it becomes less efficient than alternative routes, but the exact time for the adjustment would depend on the speed at which data is processed. It seemed a better idea to fix the cause of the saturation rather than try to compensate for it.

The second problem is more closely related to the workings of GA daemon. The main source of updates to the hop measurements comes from the ACKs generated by the dumb packets, since the fired smart packet to fired dumb packet ratio is approximately one to five and a large majority of the smart packets get lost. If we find an interesting route to a given destination and send all dumb

packets along that route, we no longer receive updates about alternative routes and eventually these routes disappear from the GA pools when their genes become obsolete

In order to solve both these problems, instead of systematically returning the best route, the daemon does a round-robin on the routes whose delay does not exceed the best delay by more than 5%.

3 GA performance evaluation

3.1 Conditions for the measurements

To evaluate the performance of the Genetic Algorithm, measurements were done on a 26 node testbed, consisting of 24 “core” machines, a source and a destination. Initially, all the machines on the testbed start with an empty dumb packet repository and empty GA pools. The network cards on the testbed were all set to 10 Mbps so that it is possible to put strain on the links without having to fire very large quantities of packets.

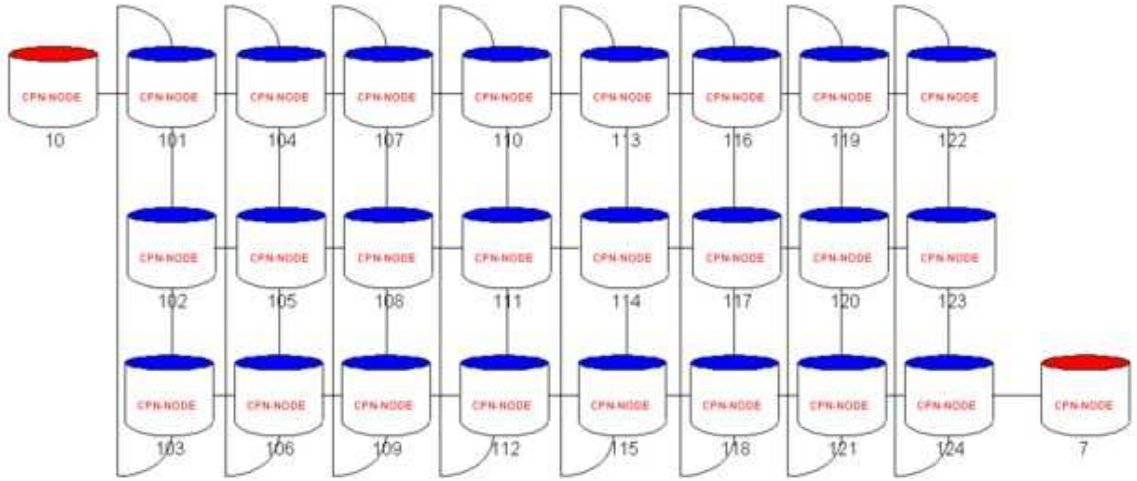

Figure 4: topology of the testbed

The GA daemon is then started up on the source node and we send 100 byte data packets from source to destination over the CPN network at a constant rate. This results in both smart and dumb packets being fired and as these packets reach there destination, acknowledgements are generated. On the source node, the CPN module is slightly modified to log packet departures and arrivals to a file, as well as the routes used and the associated delay. By parsing the log file we can determine which packets arrived and plot the information concerning them.

It is interesting to note that in a first set of experiments, the destination was not fixed but picked randomly. As has been previously stated, the GA daemon divides the paths into subpopulations based on the destination and crossover is done within a given destination. The gene pool is common to the various subpopulations, which means that we have slightly more current measurements than if we considered only one destination, but on the whole when there are several destinations the system behaves as if the destinations where considered independently with their respective packet rates. For this reason, the rotating destination was subsequently abandoned as it only complicated the result analysis without providing us with any extra information.

As currently the QoS discipline supported by the GA daemon is minimum delay, the principle means of evaluating the performance of the GA engine is by comparing the round-trip delays of smart and dumb packets with and without GA. Delay measurements and packet IDs are based on timestamps generated by the CPN module with a resolution of $10\mu s$, so all time-related measurements use this as their unit.

3.2 Smart packet paths and delays

The measurements for smart packets for GA and non-GA experiments have the same characteristics as smart packet routing is not affected by the Genetic Algorithm.

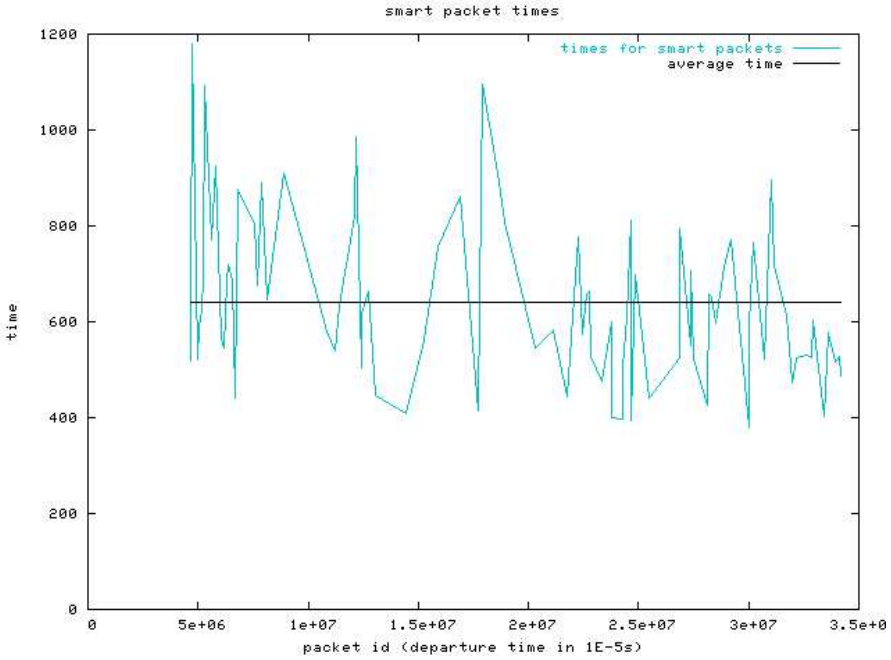

Figure 5: smart packet delays at 100 packets/s

The round-trip delay for smart packets is random, and smart packets use paths of variable lengths, which reflects the smart packets' function, that is discovering new paths. Also, the majority of smart packets are lost as they are dropped when they reach a maximum hop count in order to avoid packets running forever on the network.

When the results are parsed, paths are assigned increasing tag numbers so that it is possible to track path changes over time. Once again the smart packets' scouting function is visible as they are constantly bringing back new paths.

In the following experiments only the average delay for smart packets will be stated as it gives some information about the strain being put on the network.

3.3 Dumb packet measurements

3.3.1 Packet rates up to 200 packet/s

When using rates of two hundred packets per second or less, the load put on the network is low meaning there is a linear relation between packet delays and path lengths. The best performance in this scenario would therefore be achieved with a simple shortest route discipline, and the shortest paths from source to destination have eleven nodes.

In CPN without GA, whenever a new route to a given destination is brought back by a smart packet, this route is put into the dumb packet repository regardless of the performance of this route. This means that although short paths are discovered, they are replaced by whatever path is discovered next, producing a characteristic step pattern in the dumb packet round-trip delays.

On the other hand, when GA is enabled, as low-delays paths are acquired, they continue to be used. It is also possible to see the round-robin being performed by the GA daemon both before

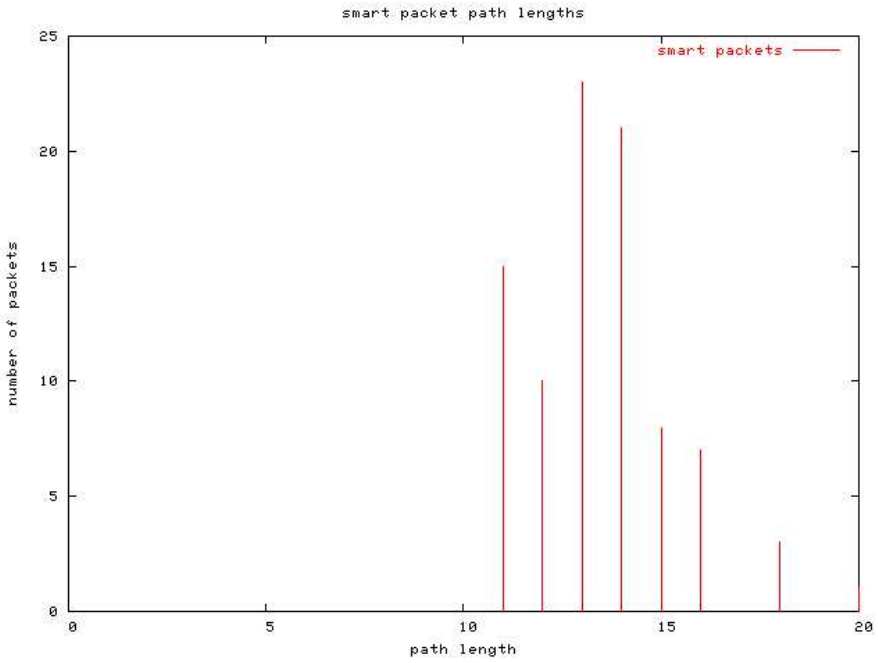

Figure 6: smart packet path lengths at 100 packets/s

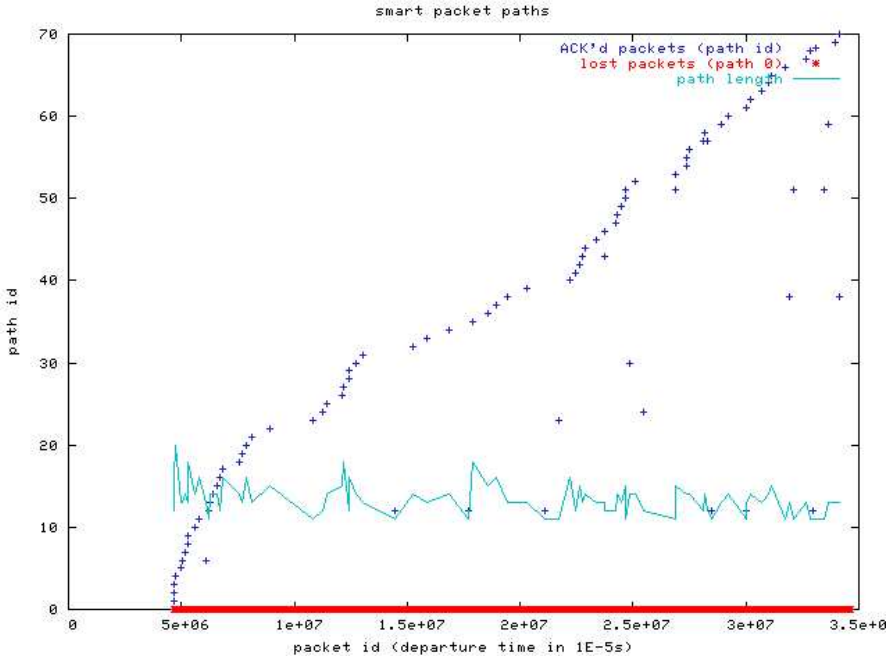

Figure 7: smart packet paths at 100 packets/s

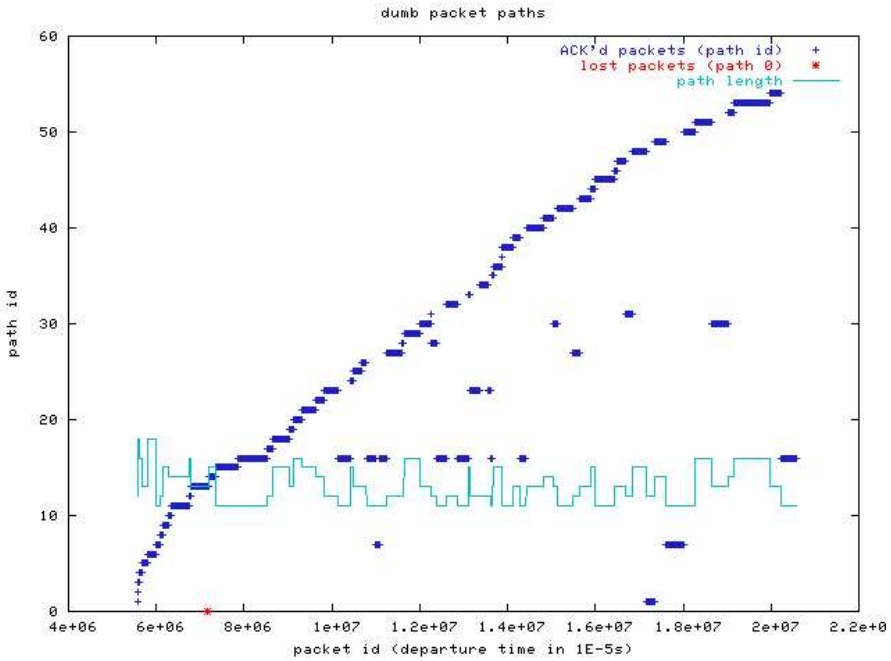

Figure 8: dumb packet paths at 200 packets/s (GA disabled)

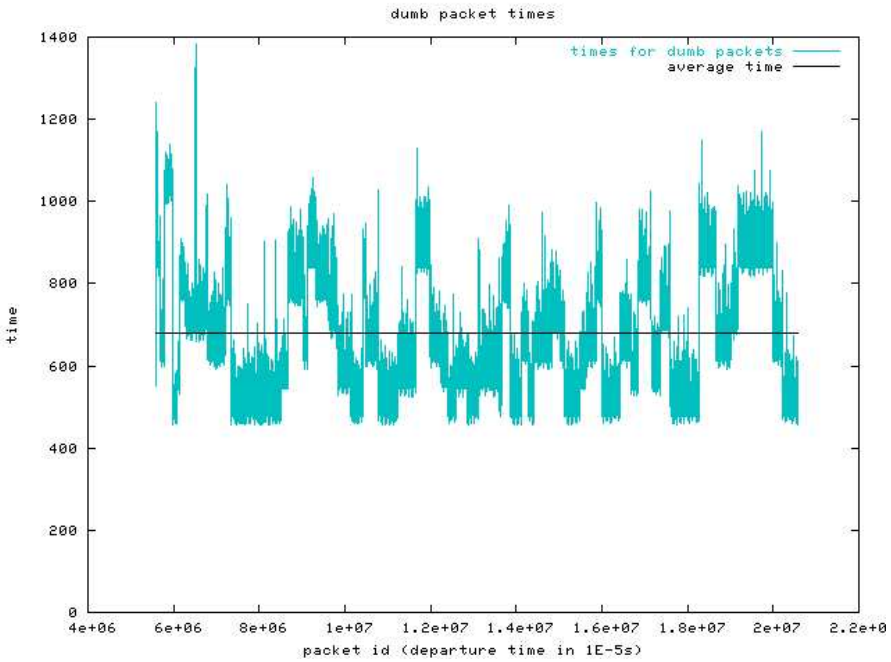

Figure 9: dumb packet times at 200 packets/s (GA disabled)

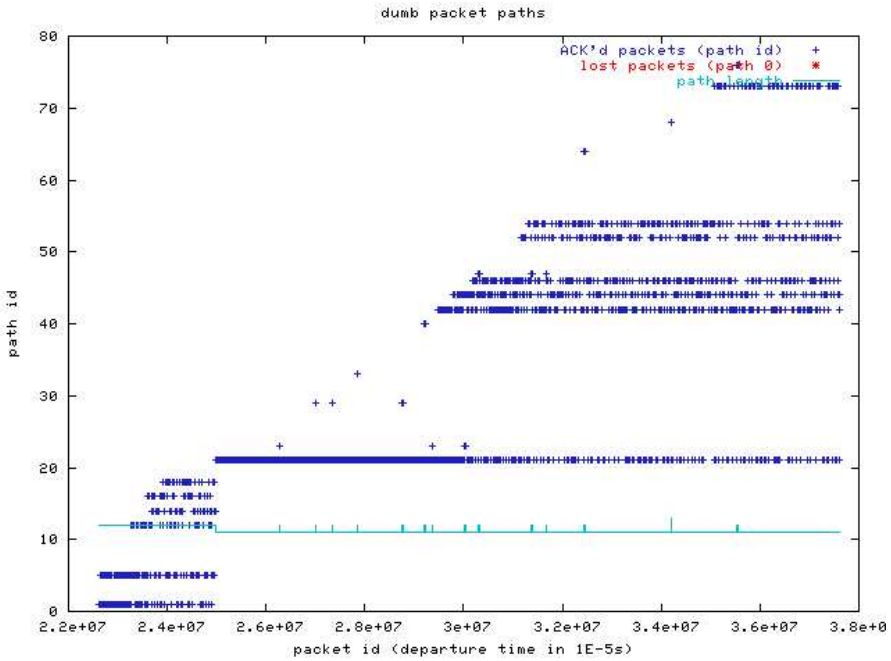

Figure 10: dumb packet paths at 200 packets/s (GA enabled)

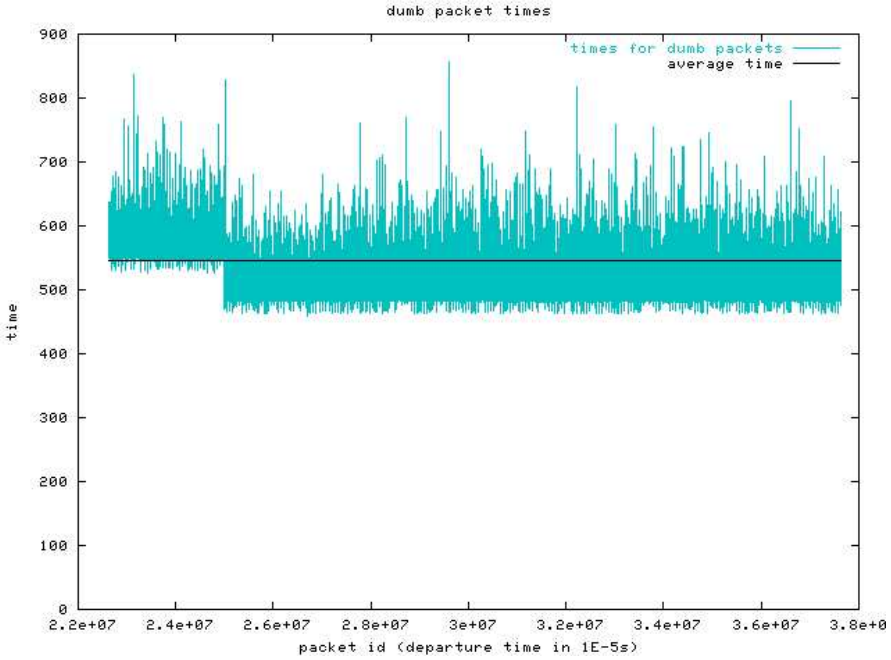

Figure 11: dumb packet times at 200 packets/s (GA enabled)

timestamp 2.5e+7 and after timestamp 2.9e+7. At timestamp 2.5e+7 the GA daemon discovers an eleven node path and abandons the round-robin on twelve node paths, but as soon as it has some alternative eleven node paths it resumes the round-robin policy on the newly discovered paths. As one might expect, the round-trip delay for dumb packets is more consistent when the GA daemon is activated.

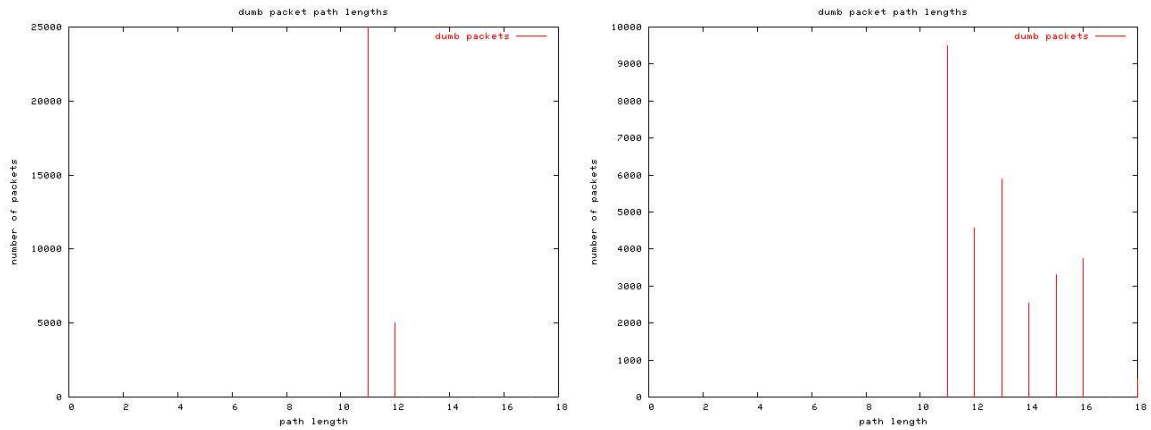

Figure 12: dumb packet path lengths with and without GA at 200 packets/s

In this scenario, GA outperforms regular CPN with an average round-trip delay of 54ms versus 68ms for regular CPN. The gain at this low packet input rate is due nearly exclusively to the crossover and selection operations, the load balancing introduced by the round-robin having little effect as the load is low.

3.3.2 Packet rates above 200 packets/s

If we increase the input rate above 200 packets/s, the delays on the network rise, and delay values are no longer proportional to the path length. The most efficient discipline is no longer systematically shortest path. We continue to observe different behaviours when the GA daemon is running and when it is not running.

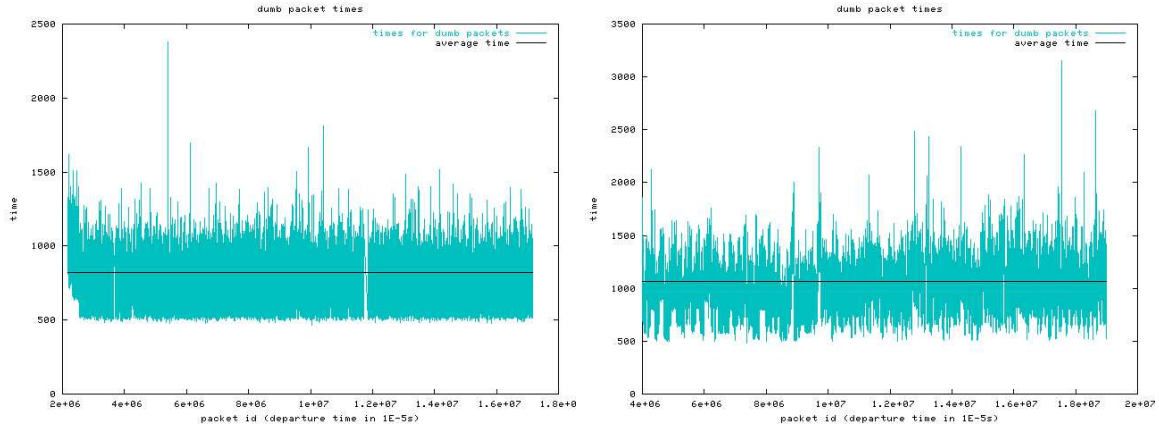

Figure 13: dumb packet delays with and without GA at 600 packets/s

Just as CPN assisted by the Genetic Algorithm yielded shorter delays than plain CPN at low input rates, it continues to do so at higher packet rates. It is no longer possible to visualise the jitter

	GA enabled	GA disabled
average roundtrip delay	82 ms	106 ms
standard deviation	15 ms	22 ms

on the delay straight from the plots, but the standard deviation on dumb packet round-trip delays is considerably lower when the GA daemon is running.

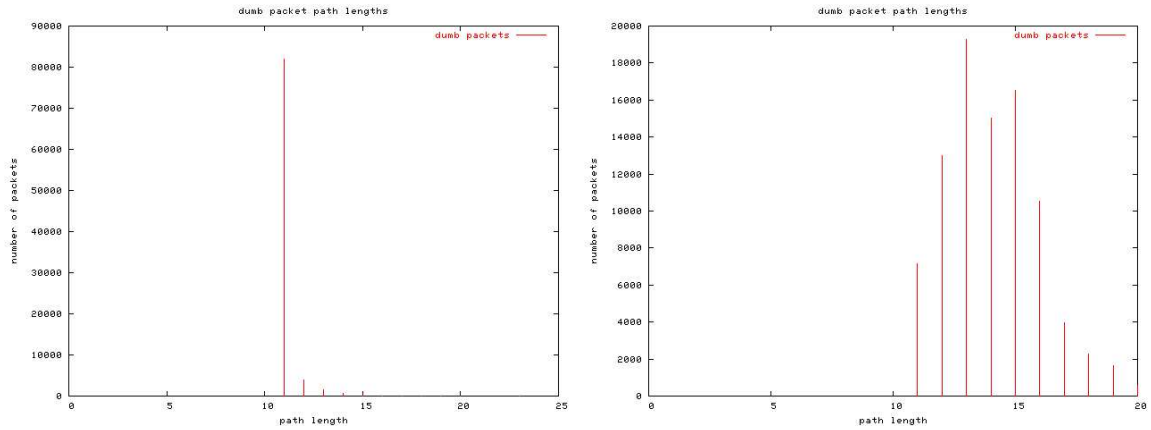

Figure 14: dumb packet path lengths with and without GA at 600 packets/s

As in the low packet rate scenario, when the GA daemon is active the vast majority of the paths that are used are of minimal length. However the increase in the packet rate introduces greater differences in the delays of the various hops on the network and so longer paths are sometimes used as they become more efficient than the minimal length paths.

3.3.3 Overview of GA and non-GA performance

The highest rate that was successfully tested on the testbed was 800 packets/s, the limiting factor being the speed at which the system's logging daemon can write to disk. In the 800 packets/s results disruptions in the logging start to appear, showing up as periods where there are no more measurements.

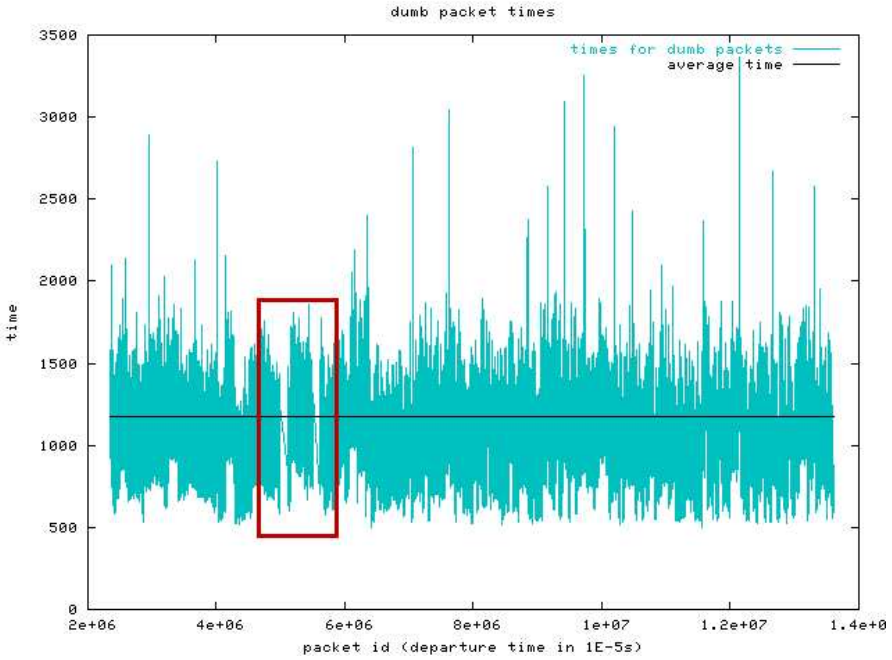

Figure 15: disruptions in a measurements at 800 packets/s (GA disabled)

The graph shows the evolution of the average round-trip for dumb packets with and without GA looks is represented for rates of up to 800 packets/s.

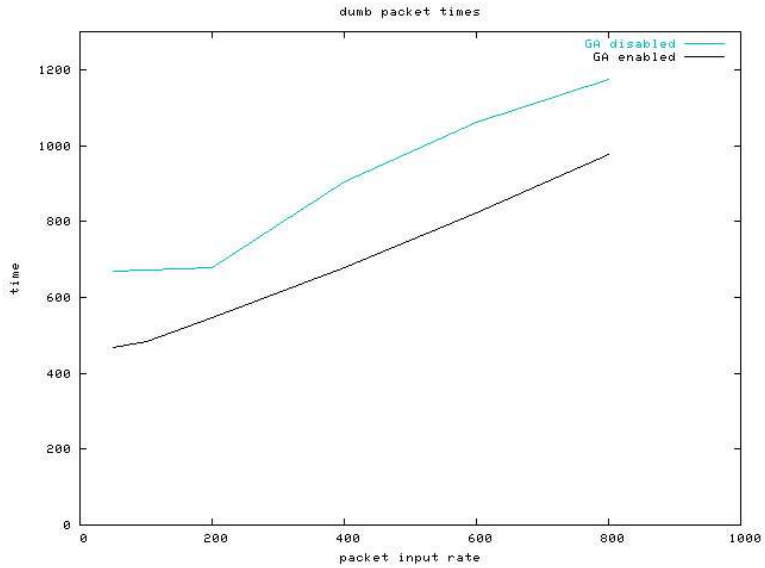

Figure 16: average round-trip delays with and without GA

3.4 System load

When the network is idle, the GA daemon uses a total of 376kB of memory, 316kB of which is in fact shared memory used by standard libraries. When firing 800 packets/s on the 26 node testbed, the peak memory usage was 384kB, which fell back to the original value once the experiment had ended and all the genes of the gene pool had become obsolete.

As for processor usage it is difficult to give an exact figure as after running for an hour with a firing rate of 800 packets/s the cumulated processor time used by the daemon was below a second. In any case, the GA daemon complies with the requirement for a low system load.

Conclusion

Building upon the existing Cognitive Packet Networking module and running only at the source node, the Genetic Algorithm daemon significantly improves performance when the quality of service discipline is minimum delay. Through successive crossover and selection operations, the GA daemon enables the source node both to compose new routes from existing ones and to select which routes to use based on their fitness. Furthermore, a round-robin policy on the best routes acts as a simple load balancing system which is desirable at high packet rates.

An interesting line of future research would be to interface the daemon with the loss measurement code for CPN which is currently being developed. With the daemon architecture and the GA operations in place this would require only minimal changes to daemon and CPN module code and would considerably broaden the scope of the resulting system.