UNIVERSITY OF AMSTERDAM

UvA-DARE (Digital Academic Repository)

Integrating networks with Mathematica

Strijkers, R.J.; Meijer, R.J.

Publication date
2008

Published in
9th International Mathematica Symposium 2008: Electronic proceedings

Link to publication

Citation for published version (APA):
Strijkers, R. J., & Meijer, R. J. (2008). Integrating networks with Mathematica. In *9th International Mathematica Symposium 2008: Electronic proceedings* (pp. 1-10). Technische Universiteit Eindhoven (TU/e). http://bmiaserver.bmt.tue.nl/eProceedings/WWW/IMS_2008_e-Proceedings.html

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: https://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.

UvA-DARE is a service provided by the library of the University of Amsterdam (https://dare.uva.nl)

Download date:09 Mar 2023

Integrating Networks with *Mathematica*

Rudolf J. Strijkers

Robert J. Meijer

University of Amsterdam, Amsterdam, The Netherlands
TNO Informatie- en Communicatietechnologie, Delft, The Netherlands
strijkers@uva.nl, robert.meijer@tno.nl

We have developed a concept that considers network behavior as a collection of software objects, which can be used or modified in computer programs. The interfaces of these software objects are exposed as web services and enable applications to analyze and manipulate networks, e.g. to find articulation vertices or configure non-default routes. This article describes the software that allows Mathematica to optimize networks and applications in a continuous cycle of monitoring, analysis and adaptation. Here, the full collection of Mathematica algorithms becomes available to calculate the next optimal configuration. These algorithms use parameters such as energy consumption, application performance or bandwidth consumption as input. As a result, Mathematica can automatically adapt the network to maintain or provide (sub-) optimal or 'better-than-best-effort' services. Furthermore, integration of networks with Mathematica allows a multi-scale optimization approach, where local behavior is adapted to support global goals. The presented work makes extensive use of Mathematica's Dynamics and Combinatorica Package, but also of web services and J/Link.

Introduction

Next-generation networks open up service interfaces of individual network elements to allow distributed and networked applications more control over network behavior. The foremost reason to allow more control is that the resource consumption of these applications should be in balance with the resources networks can provide. Any imbalance directly impacts network performance and perceived quality of experience (QoE). Typical examples that rely on good QoE include VoIP and video streaming, but also communication intensive supercomputing tasks. Therefore, networks need to orchestrate traffic to achieve 'better than best-effort' services, i.e. optimize resource utilization to meet robustness or efficiency requirements.

The need to have more control over network behavior can also be found in recent developments in photonic and hybrid networks [1]. Photonic switches connect ports to form light paths and have no knowledge of traffic flows. One of the main issues is how to orchestrate the configuration of photonic paths according to traffic flows that enter the network. This issue motivates research in multi-scale optimization, where technologies at different layers in the OSI reference model have to cooperate to fully utilize the network infrastructure.

Our research focuses on the development of architectures and models that facilitate multi-scale adaptation and 'better than best-effort' services. In [2], we presented User Programmable Virtualized Networks (UPVN), an architectural framework that enables interworking between applications and networks. In this framework, application-specific network services can be implemented in the form of component-based software objects, potentially modifying or adding behavior at different layers of the OSI reference model. These objects are typically accessible through web service interfaces and can be included as part of distributed and networked programs. Using these objects, such programs can orchestrate network resources to meet their needs. Furthermore, the framework allows not only applications to orchestrate network resources, but also enables the network to run self-adaptation programs.

Mathematica's environment is well suited for solving the, often combinatorial, problems in networks. In cases where combinatorial solutions are unfeasible or not possible due to the unpredictable nature of the network, trial and error algorithms can be used. However, trial and error algorithms require middleware support to maintain front-end interactivity with the Mathematica kernel. By providing this middleware and integrating it with Mathematica and UPVNs, Mathematica programs can continuously adapt network behavior.

This article is organized in three parts. The first part describes the use of web services to facilitate the integration between network services and applications in current state-of-the-art networks and utility computing environments. It also presents the concepts and implementation of UPVN and describes the software architecture for integration with Mathematica. The second

part demonstrates experimental results by example and the last part reflects on the results and provides directions for future work.

Web Services in Telecommunications and Virtual Operating Systems

Service Oriented Architectures (SOA) and web services as its implementation technology is becoming increasingly common in networks [3]. The devices within the network expose their functions as web services and by combining these; new, application-specific services can be composed. Web services enable high flexibility in network infrastructures and give the operator the ability to adapt services when needed. The usefulness of this approach can be observed in the many web service based deployments of large-scale hybrid networks that offer users the ability to control parts of the infrastructure [4, 5]. In these hybrid network deployments, however, the network functions remain static. Their primary service is to facilitate circuits between applications or computing clusters.

Web services are also becoming available in utility computing environments to orchestrate resources, the most notable examples are Xen [6] and VMware [7]. VMware allows monitoring and control of the whole spectrum of virtual and physical resources through web services. Consequently, the whole utility computing environment can be optimized by specialized applications such as Mathematica.

Software Framework

The UPVN model (Figure 1) describes architectural principles for programmable network services. UPVNs enable applications to upload code to network elements (NE) in the form of software objects called application components (AC). ACs allow implementation of not foreseen and application-specific behavior and allow computer programs to access their service interfaces through network components (NC). NCs act as proxies that provide redirection, virtualization or composition of multiple AC service interfaces. The manner in which NCs are exposed to applications such as Mathematica is application-specific, and technology choices and AC to NC interfacing depend on the network's application domain. For example, when the network is under full control of a single owner, it is not necessary to support multi-user mechanisms or a complex security infrastructure. NCs can offer AC services through a synchronous or asynchronous interface. With the synchronous interface, applications interact with UPVNs by instantiating the appropriate NCs that act as remote procedure call proxies. In the asynchronous interface, the application, network or middleware implements an event mechanism that continuously polls ACs and fires events in case of state changes.

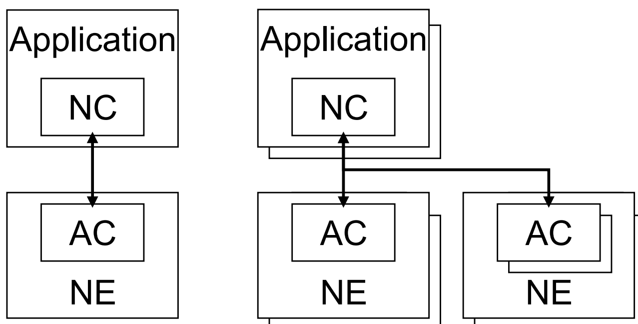


Figure 1. The UPVN model defines the elementary functions to get information from, use, and adapt network resources in computer programs.

Figure 2 shows a practical implementation architecture for integrating Mathematica in UPVNs. From Version 5 on, Mathematica supports synchronous interaction with web service enabled networks. The kernel architecture, however, does not allow simultaneous monitoring, visualization, and manipulation. Therefore, Java middleware implements an event mechanism that supports asynchronous updates. It enables the frond-end to remain responsive while providing continuous updates to the kernel. With this mechanism, a user can write a network visualization program with Dynamics in one notebook, for example, to display a continuously updated graph with marked articulation vertices while calling NC functions to manipulate the network in another.

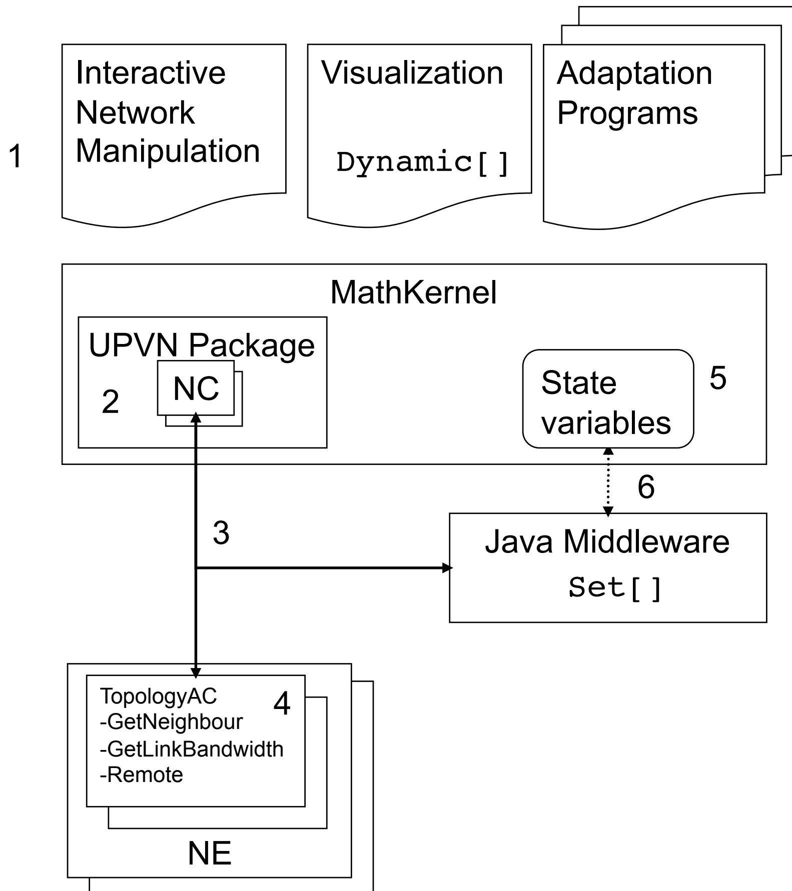


Figure 2. Notebooks (1) allow users to visualize, write and prototype programs for network manipulation or analysis in an interactive manner. The programs interact with the network through a UPVN Package (2), which implements the basic functions, such as data and type conversion to interact with Mathematica. The UPVN Package communicates through web services (3) with ACs, such as a topology AC (4). Through the same interface (2), asynchronous interaction is handled. Java middleware takes care of the updates in the MathKernel (5) through MathLink (6). Notebooks can update or programs can adapt the network in a continuous cycle with Dynamics.

We have built a test bed to conduct experiments and to validate the UPVN model [2] on a real network. In the test bed, ACs act and operate on physical resources. Because an application interact with the network through NCs, simulated or real network behavior is transparent to it. Therefore, simulating a network, which would otherwise be difficult to setup or build physically, can be useful in prototyping adaptation algorithms for example. However, as in any simulation environment care should be taken when making assumptions.

Network Modelling

For any visualization, analysis or adaptation problem there is an underlying model that transforms the measured properties into a useful representation. It is not necessary to model all the intricate details and the full scope of the network for the representation to be useful and the scope of what control is limited by what can be successfully managed. For example, a large collection of NEs can be load-balanced to macroscopic network properties such as throughput. However, when adapting NEs to service requirements of many applications at once the amount of NEs under control of one application will be relatively small.

In many adaptation problems, knowing the current topology is a primary requirement. This information can be static and provided as input, discovered or a combination of both, and the UPVN may facilitate different methods of retrieval. A basic discovery strategy consists of three parts, which may be combined to improve efficiency.

1. Discover connected NEs
2. Query NEs for neighbors

3. Construct Mathematica representation

Reachable NEs can be found by a breadth-first search. To support the breadth-first search, every NE in the UPVN test bed provides a topology AC, which implements neighbor discovery with Nmap [8]. By scanning the UPVN web services port, it also detects if a NE offers AC functions. Another topology AC function takes the discovered NEs as input and queries their IP links. When needed, a conversion routine translates the IP addresses to id's and back. With these functions, the full IP topology can be discovered and used in Mathematica. The code snippet below shows the breadth-first search code and illustrates how the network calls are embedded.

```
BFSDiscover[start_] := Module[
   {q = {}, u = start, result = {start}, vall},
   q = Append[q, u];
   While[Length[q] > 0,
    u = q[[1]];
    q = Delete[q, 1];
    vall = Remote[u, "DiscoverNetworkElements", "", auth];
    vall = ToExpression[StringReplace[vall, {"[" → "{", "]" → "}"}]];
    Do[
     If[Not[MemberQ[result, vall[[i]]]],
      result = Append[result, vall[[i]]
       ];
      q = Append[q, vall[[i]]];],
     {i, 1, Length[vall]}];
   ];
   result
  ];
```

DiscoverNetworkElements finds the neighbor UPVN NEs of the calling NE. Remote is a special function in the topology AC. It calls a remote web service from the host it executes on. This way, the AC supports a recursive search from a single NE. Note that disconnected elements will not be found. The web service calls return strings and need to be converted to the correct Mathematica data types. Type conversion can be abstracted away from the programmer in the NCs.

Once the network topology is acquired and a representation is constructed, the full range of Mathematica's environment becomes available for network visualization, analysis and adaptation. We consider two common categories for network related problem solving. The first category uses combinatorial methods. Mathematica's Combinatorica Package provides many combinatorial algorithms that can be applied without modification. The second category uses a trial and error approach. It uses historical data to converge to an (sub-) optimum and requires a continuous cycle of monitoring, analysis and adaptation. In addition, visualization of information flows and network configurations can aid in problem solving.

■ Network Manipulation

These packages are required:

```
Needs["Combinatorica`"]
Needs["GraphUtilities`"]
```

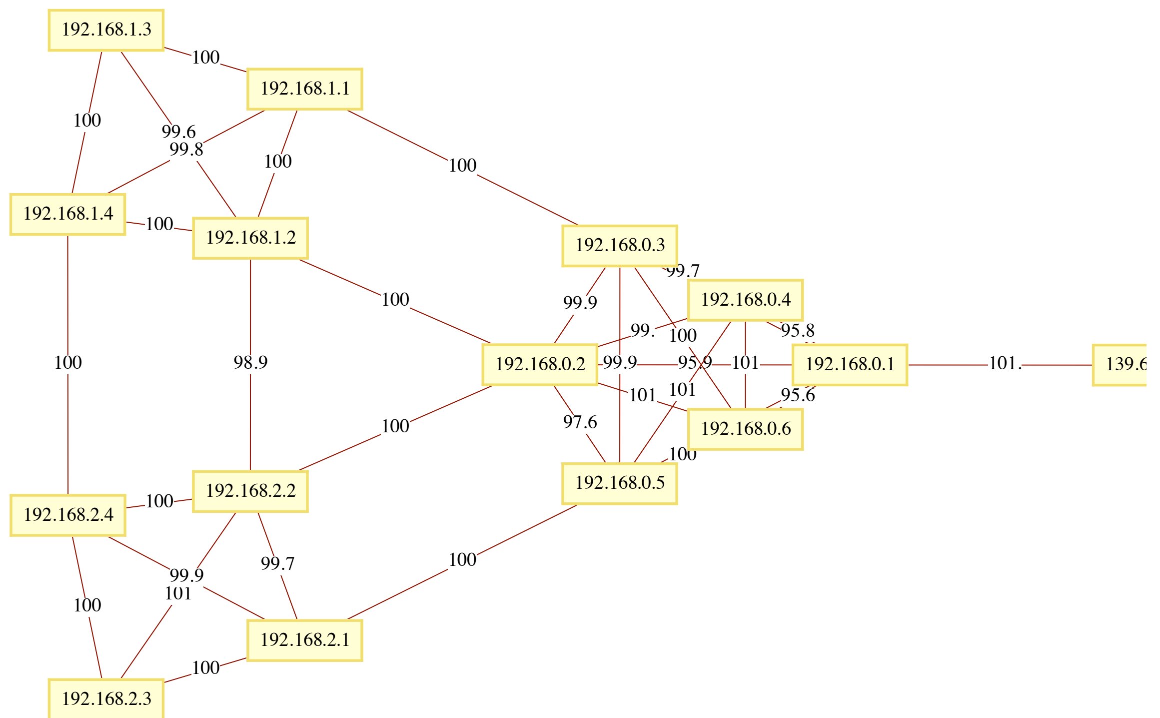


Figure 3. The right edge of the UPVN test bed connects to the Internet, and allows network discovery from any UPVN NE that knows the address of this gateway. The edge weights were obtained by performing a bandwidth probe on the discovered IP links. The topology AC uses Iperf [10] for performance measurements.

Consider the network given by Figure 3. It shows the discovered IP topology of the UPVN test bed together with results of bandwidth measurements. Now, with the aid of the Combinatorica Package it is possible to route traffic in exotic ways. For example, to broadcast a stream to all nodes in a network the minimum spanning tree can be used. The result of the calculation can be configured in the network in various ways. In the test bed, it was done by tagging application traffic and adding forwarding rules for the tags. The following code shows some examples and how forwarding rules are generated and fed to the appropriate NCs.

```
(*Convert between NE address and Mathematica representation *)
NEtable = Apply[Union, #[[1]] & /@ network];
NE2Index[nes_] := Position[NEtable, #][[1]][[1]] & /@ nes
IndexToNE[is_] := (NEtable[[#]]) & /@ is
```

In the network of Figure 4. it is trivial to find articulation vertices.

```
(* Quick conversion from NE addresses to Combinatorica graph *)
myg = #[[1]] & /@ network;
myg2i = NE2Index[#] & /@ myg;
g = EmptyGraph[Length[NEtable]];
Do[g = AddEdge[g, myg2i[[i]]], {i, 1, Length[myg2i]}];

IndexToNE[ArticulationVertices[g]]

{192.168.0.1}
```

Now generate the minimum spanning tree and show the NC calls to configure the network.

```
mst = IndexToNE[#] & /@ Edges[MinimumSpanningTree[g]];
Remote[#[[1]], "AddForwardingRule", "green:" <> #[[2]] ] & /@ mst
```

```
{Remote[139.63.145.94, AddForwardingRule, green:192.168.0.1],
 Remote[192.168.0.1, AddForwardingRule, green:192.168.0.2],
 Remote[192.168.0.1, AddForwardingRule, green:192.168.0.3],
 Remote[192.168.0.1, AddForwardingRule, green:192.168.0.4],
 Remote[192.168.0.1, AddForwardingRule, green:192.168.0.5],
 Remote[192.168.0.1, AddForwardingRule, green:192.168.0.6],
 Remote[192.168.0.2, AddForwardingRule, green:192.168.1.2],
 Remote[192.168.0.2, AddForwardingRule, green:192.168.2.2],
 Remote[192.168.1.1, AddForwardingRule, green:192.168.1.2],
 Remote[192.168.1.2, AddForwardingRule, green:192.168.1.3],
 Remote[192.168.1.2, AddForwardingRule, green:192.168.1.4],
 Remote[192.168.2.1, AddForwardingRule, green:192.168.2.2],
 Remote[192.168.2.2, AddForwardingRule, green:192.168.2.3],
 Remote[192.168.2.2, AddForwardingRule, green:192.168.2.4]}
```

When special joined paths are required, for video streaming for example, we can define a domain specific language (DSL) to specify the required paths and then join these paths if there is overlap.

```
MakeTuples[l_List] := Thread[{Part[l, 1 ;; -2], Rest[l]}]

(* This function joins paths given as a list of route statements. A
 real DSL can implement a range of algebraic path functions. *)
JoinPhts[g_, prog_, col_] := Remote [IndexToNE[{#[[1]]}][[1]],
    "AddForwardingRule", "blue: " <> IndexToNE[{#[[2]]}][[1]]] & /@
  (Flatten[MakeTuples[#] & /@ (Method[g, NE2Index[{#[[1]]}][[1]],
          NE2Index[{#[[2]]}][[1]]] /. #[[3]] & /@ prog), 1] // Union)
```

The program defines a list of routes and the method of path finding. All the paths will be collected and double edges removed. The resulting edges can be configured in the network with the same tagging facility as above.

```
JoinPhts[g, {
  Route["192.168.0.2", "192.168.2.3", Method → ShortestPath],
  Route["192.168.0.2", "192.168.2.4", Method → ShortestPath]
 }, "blue"]
```

```
{Remote[192.168.0.2, AddForwardingRule, blue: 192.168.2.2],
 Remote[192.168.2.2, AddForwardingRule, blue: 192.168.2.3],
 Remote[192.168.2.2, AddForwardingRule, blue: 192.168.2.4]}
```

The next program shows a straightforward way to find edge disjoint shortest paths [10]. The calculated paths can be configured in the same manner to provide backup paths or aggregate traffic for example.

```
RemovePath[g_, p_] := Module[{tuples = MakeTuples[p]},
  Do[g = DeleteEdge[g, tuples[[i]]], {i, 1, Length[tuples]}]; g]

AllSP[f_, s_, t_, v_: {}] := Module[{p}, If[Length[p = ShortestPath[f, s, t]] > 1,
   AllSP[RemovePath[f, p], s, t, Append[v, p]], Return[v]]]

EdgeDisjointSP[g_, src_, dst_] := Module[
  {flow, flowg},
  flow = (#[[1]]) & /@ NetworkFlow[g, src, dst, Edge];
  flowg = MakeGraph[VertexList[g],
    (MemberQ[flow, {#1, #2}] || MemberQ[flow, {#2, #1}]) &, Type → Undirected];
  AllSP[flowg, src, dst]]

IndexToNE[#] & /@
 EdgeDisjointSP[g, NE2Index[{"192.168.2.3"}][[1]], NE2Index[{"192.168.0.1"}][[1]]]
```

```
{{192.168.2.3, 192.168.2.1, 192.168.0.5, 192.168.0.1},
 {192.168.2.3, 192.168.2.2, 192.168.0.2, 192.168.0.1},
 {192.168.2.3, 192.168.2.4, 192.168.1.4, 192.168.1.1, 192.168.0.3, 192.168.0.1}}
```

■ Real-time Adaptation

The previous examples calculated the precise network configuration. In the trial and error approach the adaptations ultimately converge to a better state. This approach can be divided in three stages and correspond with an elementary feedback loop well known in control theory (Figure 4). The first stage builds a representation of the problem space by retrieving the necessary information from the network. The second stage applies a decision process to find a next better state and the third stage reflects its results by adapting the network.
This example illustrates the stages with a trial and error approach to load-balance traffic in a routed network. It introduces the idea of attractors and repulsors, which indirectly modify forwarding rules. A decision model decides where to place specific attractors or repulsors based on throughput measurements. This is an approach to load-balancing inspired by physics [11].

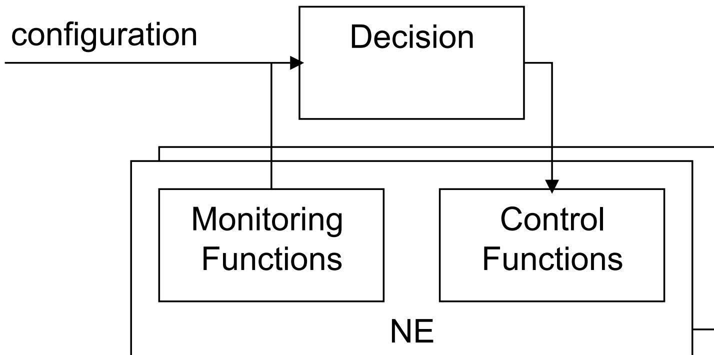


Figure 4. The trial and error approach corresponds with an elementary feedback loop. Monitoring functions and configuration options provide input to form a state representation. The result of a decision process based on the state is actuated with control functions. Monitoring and control functions can be part of different ACs.

In the Java middleware, one thread polls the network and another updates variables in Mathematica using Set[]. The following code simulates network throughput by generating two shortest paths and at each NE summing the crossing paths. The generated list is normally updated by Java.

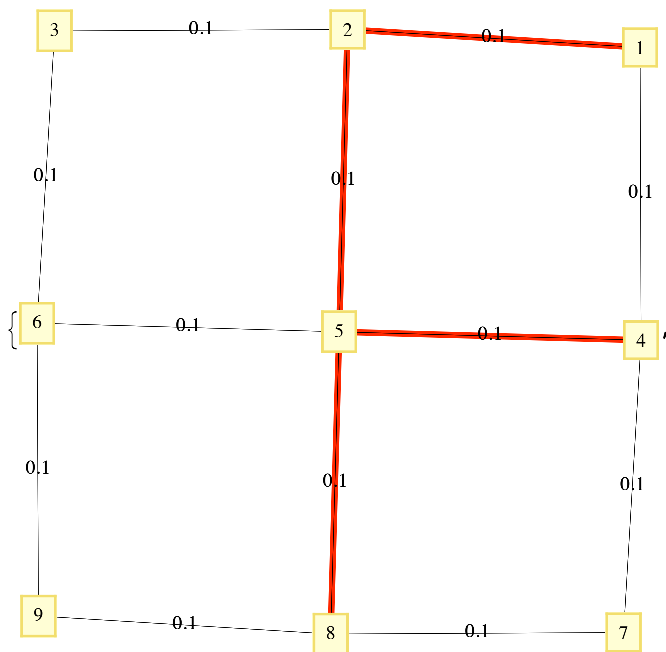
```
InitWeights[g_, val_] := SetEdgeWeights[g, Table[val, {Length[Edges[g]]}]]

g = ToCombinatoricaGraph[({#[[1]] → #[[2]], #[[2]] → #[[1]]}) & /@
      ((#[[1]] → #[[2]]) & /@ ToUnorderedPairs[GridGraph[3, 3]]) // Flatten];
g = InitWeights[g, 0.1];
paths = {{4, 5}, {1, 8}};

sp = (ShortestPath[g, #[[1]], #[[2]]]) & /@ paths;
vt = Table[0, {V[g]}]; (vt[[#]] = vt[[#]] + 1) & /@ (sp // Flatten);
vt

{1, 1, 0, 1, 2, 0, 0, 1, 0}
```

The decision model is straightforward. It finds the NE with the maximum throughput that will be repulsed.

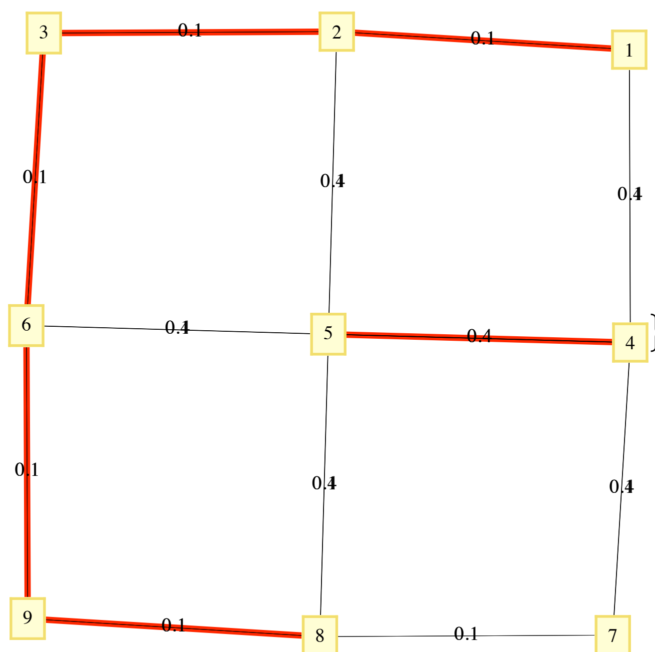
```
pos = Position[vt, Max[vt]] // Flatten

{5}
```

The following code puts it all together and the result is visualized in Figure 5. The first part of the code is responsible for the simulated input. After each edge weight update in the network, the new paths are calculated. This is analogue to changing edge weights in an OSPF routed network. Next there needs to be a stopping criteria to decide if the update had a positive effect. The most straightforward manner is to update the network a fixed number of times and pick the best result, which in the case of load-balancing is when the difference in throughput is minimal. Finally, the edge weights are manipulated by placing the repulsor and the state needs to be updated again.

```
g2 = g;
vl = {};
For[i = 0, i < 10, i++,
  sp = (ShortestPath[g2, #[[1]], #[[2]]]) & /@ paths;
  vt = Table[0, {V[g2]}]; (vt[[#]] = vt[[#]] + 1) & /@ (sp // Flatten);
  pos = Position[vt, Max[vt]] // Flatten;

  (*Calculate variance. This is the measure for a good
   solution. Small differences in NE throughputs give a small variance.*)
  vl = Append[vl, Variance[vt]];

  x = {};
  If[Max[vt] >= Round[Mean[vt]] * 2,
   (If[Length[Position[pos, #[[1]]]] > 0, x = Append[x, #]]) & /@ Edges[g2];
   g2 = SetEdgeWeights[g2, x, (# + 0.1) & /@ GetEdgeWeights[g2, x]];
  ];
 ];
```

{

,



}

Figure 5. The first picture shows theititial state, and the second picture shows the state after ten iterations. The area around NE four and five have become unattractive enough to change the path of (1,8).

It is out of the scope of this paper to provide a full analysis of this approach. However, it does demonstrate that network modelling can almost entirely be done in *Mathematica* and use ACs for monitoring and control only.

Discussion and Future Work

When networks support the elementary functions of the UPVN framework, users can develop programs to change and optimize network behavior. Network services are implemented as software objects that expose their interfaces as web services, which can be consumed to develop new or modify existing network behavior. In this article, the emphasis was on the communication between the network and Mathematica.

In our experiments, we have found three limiting factors in centralized network adaptation. The first limitations are the scope and time-scale of adaptations. In the UPVN test bed, a single Mathematica instance controls network behavior. Since in practice communication between ACs and Mathematica was in the range of seconds, it puts a bound to what can be monitored and adapted simultaneously. Third, to support real-time interactions with Mathematica, the J/Link interface provided a work-around by offloading complex real-time network communication from the kernel. It remains to be seen if this approach allows large and frequent updates and how this affects the front-end responsiveness. Tests upon now were successful in update cycles of seconds, which is already good enough to address many problems.

The next step is to embed Mathematica in NEs and explore distributed adaptation algorithms. It allows Mathematica to orchestrate the network at different levels and scopes and allow the development of adaptation hierarchies each with their own adaptation algorithm. The question here is: what if network adaptation programs are distributed over the network?

The presented concepts and software framework are currently being applied in two applications. The first application integrates Mathematica with StarPlane's middleware for light path visualization and control. Using our software framework, Mathematica can proactively switch light paths based on traffic measurements at the edges of a computing cluster. Shortest edge disjoint paths calculated from Mathematica will be used to switch and aggregate additional paths when clusters overload one path. The second application uses Mathematica to visualize, analyze and automatically adapt a VMware infrastructure. In a continuous loop, a linear program will calculate the optimal placement of VMwares considering their power consumption and adapt the VMware infrastructure to reflect the new optimal state.

Conclusion

Mathematica’s powerful symbolic programming language, large collection of algorithms and its ability to interface with Java and web services makes it a suitable research tool for the development of ‘better-than-best-effort’ network services. By utilizing UPVNs, Mathematica programs can move applications, reprioritize traffic or reroute connections to achieve optimal network performance in a single integrated environment. This aspect enables interaction between different adaptation programs and opens the way to multi-scale network optimization.

References

[1] C. Riziotis and A. V. Vasilakos, "Computational intelligence in photonics technology and optical networks: A survey and future perspectives", Information Sciences, vol. 177, pp. 5292-5315, 2007.

[2] R. J. Meijer, R. J. Strijkers, L. Gommans, and C. de Laat, "User Programmable Virtualized Networks", in Proceedings of the Second IEEE International Conference on e-Science and Grid Computing: IEEE Computer Society, 2006.

[3] D. Griffin and D. Pesch, "A survey on web services in telecommunications", IEEE Communications Magazine, vol. 45, pp. 28-+, Jul 2007.

[4] E. Grasa, G. Junyent, S. Figuerola, A. Lopez, and M. Savoie, "UCLPv2: A network virtualization framework built on web services", IEEE Communications Magazine, vol. 46, pp. 126-134, Mar 2008.

[5] P. Grosso, L. Xu, J. P. Velders, and C. de Laat, "StarPlane - a national dynamic photonic network controlled by grid applications", Internet Research: Electronic Networking Applications and Policy, vol. 17, pp. 546-553, 2007.

[6] Xen Hypervisor, http://xen.org (accessed at 20 April 2008)

[7] VMWare, http://www.vmware.com (accessed at 2 August 2007)

[8] Network Mapper, http://nmap.org (accessed at 7 April 2008)

[9] Iperf, http://dast.nlanr.net/Projects/Iperf/ (accessed at 20 April 2008)

[10] J. W. Suurballe, "Disjoint paths in a network", Networks, vol. 4, pp. 125-145, 1974.

[11] S. Toumpis, "Mother nature knows best: A survey of recent results on
wireless networks based on analogies with physics", Computer Networks, vol. 52, pp. 360-383, 2008.

Rudolf J. Strijkers (1981) received a BSc and MSc degree in computer science from the University of Amsterdam. He currently pursues a PhD degree at the same university and works for the Dutch institute for applied scientific research TNO. His scientific interests are in adaptive, programmable and intelligent networks. He uses Mathematica to implement and test new concepts and ideas.

Robert J. Meijer (1959) received a PhD degree in experimental nuclear physics of the University of Utrecht. He switched to ICT research at TNO in 1991 and became professor at the University of Amsterdam in 2002. His scientific interests are next generation networks and computer technologies with application-specific dynamic adaptation capabilities.