Modeling, Quantifying and Testing Complex Aggregate Service Chains

Carl Anderson[1], Joseph A. Rothermich[2], Eric Bonabeau[3]
Icosystem Corporation, 10 Fawcett Street, Cambridge, MA 02138
eric@icosystem.com

Abstract

Service chaining, the act of stringing a sequence of services together to form a new service, is a key element of web services. However, for web services to reach its full potential the issue of testing service-chaining at the network level must be resolved. How can one map the micro-level service-service interactions to the macro-level system performance? For instance, as service chains become longer and more complex, how do they affect end-user quality of service? Focusing on aggregate service chains—situations in which the user invokes a service that carries out the chain, without the user being aware of the individual services—we tackle these questions using a Java simulation tool to model service chaining, visualize network traffic and quantify service chain complexity. We demonstrate that one can orchestrate very complex service chains in a simple distributed manner and quantify how service chain complexity affects end-user quality of service and network loading.

1. Introduction

Web Services is set to be a huge market, a predicted $43bn in 2010 [1], but web services *testing* is a stumbling block. While practitioners currently focus on particular low-level service-service interactions, the network-level perspective is generally overlooked. This relative lack of network-level testing is especially true of the prime web service feature that will pave the way for fundamentally new (B2B) business models and data usage: *service chaining*—the act of stringing a sequence of services together to form a new service. It is a key element of web services and perhaps the best way to maximize the potential of extant resources. However, the implications of service chaining upon network dynamics and performance is far from clear, if indeed the particular chain does work as planned. In this study, we develop simulation tools to tackle this issue of network-level testing of service chains.

A service chain is formally defined as "a sequence of services where, for each adjacent pair of services, occurrences of the first action is necessary for the occurrence of the second action" [2,3]. ISO 19119 [2–6] defines three types of service chaining:

- ***User-defined*** (transparent): the human user manages the flow;
- ***Workflow-managed*** (translucent): the human user invokes a workflow management service that controls the chain and the user is aware of the individual services;
- ***Aggregate*** (opaque): the user invokes a service that carries out the chain, with the user having no awareness of the individual services.

Aggregate service chains, the focus of this study, are effectively a double-edged sword: on the one hand, they i) represent the greatest ease-of-use for non-expert end-users—users need not know how the service performs its advertised functionality, only what it does and how to use it—and ii) these chain types also represent the greatest *practical* modularity. That is, a service chain is also a service *per se*, and so, in turn, it can be chained with other services to provide new services and new functionality. However, such chains can very quickly become unwieldy, overwhelming and unmanageable if the user must coordinate the chains themselves, whereas the self-contained modularity of aggregate service chains can render such complex chaining feasible.

On the other hand, this opacity may also be its downfall. Chaining services, which themselves may be service chains, may lead to a service which when invoked sets off a huge cascade of requests and data transfer across

[1] Current address: Qbit LLC, 6905 Rockledge Drive, 3rd Floor, Bethesda, MD 20817. Email: eggandchips@gmail.com
[2] Current address: Natural Selection, Inc., 3333 N. Torrey Pines Ct., Suite 200, La Jolla, CA 92037. Email: jrothermich@natural-selection.com

[3] Corresponding author.

the network—in short, the user may simply be unaware of the complexity of some "mega-chain" that they have created or invoked. In addition, when chaining service *chains*, one must ensure that the interdependencies introduced into the chain do not create some pathological race condition. This either requires some global overview of the interdependencies among a suite of services or some more decentralized loop-checking algorithms, which will inevitably generate even more traffic across the network, thereby exacerbating any detrimental effects of such complex chains.

If web services is to reach its full potential the issue of testing service chaining *at the network level* must be resolved. How can one link and map the micro-level service-service interactions to the macro-level system performance? For instance, as service chains become longer and more complex, how do they affect end-user quality of service?

In this study, we tackle precisely these questions using a Java simulation tool to model service chaining, visualize network traffic and quantify service chain complexity. Importantly, we assume that all low-level individual service-service interactions work perfectly—our focus is not only on service chain but on more complex service chains than are generally possible today. We wish to explore issues that the web services community will face in the next few years when service chains become more complex. We develop a simple grammar that defines a single service and individual links in a service chain and use this to generate random service chains. We also

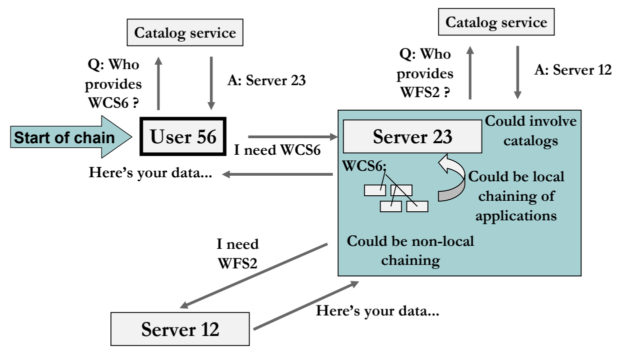


Figure 1: An overview of the functionality implemented in this web services model: User 56 uses a catalog to find a provider of service WCS6 and then submits the request to server 23. Server 23 begins to process WCS6 but part of this service involves a call to service WFS2. The server does not host this service and so uses a catalog to find a service provider: server 12. Server 23 submits the request, which is processed by server 12 who returns the data to server 23. Eventually, server 23 completes the service and returns the data to the user.

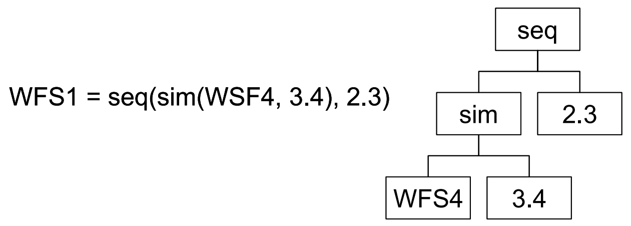


Figure 2: a subservice definition for WFS1, shown in as a LISP-like expression (left) and expanded as a tree (right). This subservice contains both sequential and simultaneous processes. To complete WFS1, both 3.4 time units of local processing and a call to subservice WFS4—thus making it a service chain—occur simultaneously and independently (sim node). Only when both of these subtasks have finished (seq node), can 2.3 time units of local processing be performed to complete the task.

develop an objective metric (which is sufficiently generic to be used with any service chain) to calculate the "hierarchical process complexity" [7] of a service chain. As such, we are able to investigate the relationship between service chain complexity and end-user quality of service (QoS), our major focus in this paper.

2. Web Services Model

`In brief, we adopt an agent-based [8] approach to model a set of generic end users that submit service requests to a network of generic machines that host those services. Many of the services call upon other services to complete their task and so constitute service chains. Both users and servers make use of catalog (registry) services where necessary to provide a service provider. An example of a simple service chain is given in Figure 1 in which User56 submits a request for service WCS6 (which in turn calls WFS2) to server 23.

2.1 Services and subservices

The atomic unit of a (sub)service is some local processing of data or a file I/O operation. In the model, this is represented solely as some duration, irrespective of the particular operation. These atomic processing units are combined with different temporal dependencies—simultaneously (`sim`) and sequentially (`seq`), detailed below—to form a service. However, these services may form individual links and components of a service chain and hence are *sub*services from a service chain's perspective.

A (sub)service is defined by its LISP-like expression, conceptually equivalent to a script, algorithm, or application that defines and implements the operations and data processing necessary to complete that (sub)service. (Such expressions can easily be ported to XML, BPEL4WS, DAML-S and the like [9].)

The subservice grammar is simple: nodes are either durations (terminal leaves) or a `sim` or a `seq` node. `seq`(X,Y) means that X, whether it be local processing (i.e. a duration) or a call to a (sub)service, must complete before work on Y can commence. Conversely, `sim`(X,Y) means that both X and Y can be worked on simultaneously, e.g. if both X and Y are calls to other services. In this implementation, `sim`() may take 2 or more arguments and `seq`() just two arguments (but which can be nested; e.g., do X then do Y then do Z: `seq(seq(`X,Y`),`Z`)`).

An example subservice, WFS1, is shown in Figure 2. To complete the service, the server must send a request to WFS4—i.e., another subservice, thus making WFS1 a service chain—and do some local processing that will take 3.4 time units. The server can do these two actions simultaneously (as defined by the `sim` node). Then, as defined by the `seq` node, once the WFS4 and 3.4 time units have been completed, the 2.3 units of local processing can start. After all of the tasks in WFS1 have been performed, the subservice is complete and can be returned to its requestor: a user, another server, or itself. Notice that WFS4 describes another service to be performed (which may be a chain itself). This subservice may be available on the current server; if not, however, a new request would be created by the server and sent off to some other server that hosts that subservice.

We loosely base our examples upon the Open Geospatial Consortium's (OGC) Open Web Services Thread Set 2 (OWS-2) [10,11]. We model the following services only: catalog service, web coverage service (WCS), web feature service (WFS) and web mapping service (WMS) which define a high level functionality. Within each of these four services, however, are a number of specific subservices that a server may host, say WCS1, WCS2,…,WCSn. (Each server has a constant probability of hosting each service, and then a constant probability of providing each subservice of any service that they host.) This allows us to capture the essence of a set of servers that may specialize on a set of subservice types (e.g., a NASA server that primarily provides WCS subservices). However, the (randomly-generated) subservice definitions used in the model can be considered completely generic.

In the simulation model, subservices (and consequently service chains) are created randomly. That is, the number of leaves is either constant or first chosen from a discrete uniform distribution. Then, randomly using `sim` and `seq` nodes, a tree is generated with that number of leaves. Finally, with a certain probability, p_{leaf}, each leaf is designated a duration or as a subservice call (such as WFS4 in Figure 2). Durations are generated from some uniform distribution.

To avoid pathological dependency cycles (e.g., service A calls and depends upon service B which, in turn, calls and depends upon service A), all of the subservices are placed in a "pecking order" (i.e., some assigned hierarchical level) in they may only call upon on services lower in the hierarchy. That is, for n subservices, we create a "ladder" with $\lfloor\sqrt{n}\rfloor$ rungs and randomly assign subservices to a rung. Subservices may only call upon services on a lower rung and those on the lowest rung may not chain: their leaves must be durations only (but with the same tree structure).

2.2 Users

We model a set of end users, each of which has a "profile": a set of service types which they may request. Users submit a Poisson stream of independent service requests to the network at rate λ (this rate varies among users: $1 / \lambda \sim N(500,2)$).

Users will choose a new request type from their profile (with some probability) otherwise they will choose a random service from a catalog. If they choose from a catalog, this service type is added to their profile and so over time their profile grows, simulating "exploration."

Each request type in the profile is associated with a relative probability of being chosen (given that the user has decided to submit a new request). After having chosen a request type, the user will use a catalog service, if necessary, to find a provider for that service type. However, users possess a (infinite) memory and will remember service providers that they have previously submitted certain request types to.

Users keep a log of the times at which each service request is submitted and returned. This is the basis of user the quality of service metrics: the mean and median time to complete a request (detailed later).

2.3 Servers

Servers are generic, single-CPU computers that host and provide services and files. When the CPU updates, new requests in the "inbox" are placed at the end of the job queue whereas returned data (from a previous request sent out) are used to update jobs in the queue that are waiting for that data. The CPU polling scheme is first-in-first-out with respect to those jobs that can actively be worked on at that time. That is, suppose the first job to arrive is `seq`(X,Y) where X is a service request to another machine and the second job to arrive is `seq`(3,*Z*). The CPU will first process the request for *X* (using catalogs to find a service provider if necessary) but then *Y* cannot be worked on until *X*'s results return. Thus, the CPU will move to job 2 and work on its 3 time units of local processing. Whether *Z* gets processed before *Y* entirely depends upon *X*'s return: within 3 time units, process *Y* otherwise process *Z*. The CPU updates—clears its inbox—when it receives a high priority interrupt service request (not implemented in these results), completes a job or after a default interval, whichever is the sooner.

2.4 Visualization

To visualize the user-server network and its web service traffic, we extended the Java TouchGraph library [12] that provides a spring graph layout for a set of nodes; here, individual servers and users. Links (the gray lines in figure 3) either represent a current inter-server request or they are user-server links, representing the servers that a user *may* submits request to, given their profile. Envelopes that move along the links represent new (white) or complete (gray) requests. Details on the server nodes provide information on job queue length (using a bar chart) and a high level listing the OWS services that the server provides (popup menus provide a full listing, will complete definitions, of the services). User nodes list the number of outstanding requests. Such a visualization tool was found to be very useful in both debugging the system and also for identifying regions of the network that had become overloaded.

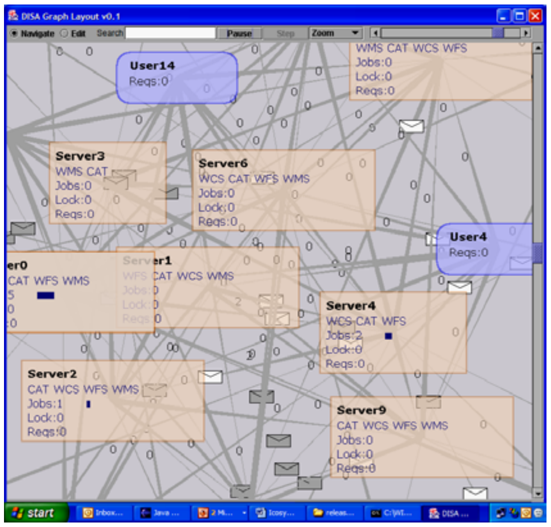


Figure 3: a detail of part of a web services network using the TouchGraph library (see text).

2.5 Service chain complexity metric

In this study, we wished to explore the effect of service chain structure upon network dynamics and quality of service metrics. How though does one quantify a service chain? Clearly, we wish to take into account the number of elements in the chain: a linear chain of six elements requires more work and coordination than a chain of three elements. However, how can one incorporate the other major feature of service chains: their temporal dependencies? We need to differentiate between `sim`(*X*,*Y*,*Z*) and `seq`(`sim`(*X*,*Y*),*Z*) which include the same components but which have a very different chain structure. In summary, in terms of number of elements, longer service chains are clearly more "complex" than shorter chains, and in terms of dependencies, branching chains seem more "complex" than linear chains in some intuitive but hard-to-define sense.

We used a complexity metric that captures both of these aspects objectively. The metric was originally developed to quantify the complexity of the hierarchical structure of cooperative tasks in insect societies [13]. The logic of the metric is as follows: it starts from the top of the structure—the start of the service chain—and considers the relationship between any directly dependent subunits: "adjacent" services. If the node is a branching point, it is a given more points than if it constitutes a single linear link to the next node in the chain because it requires a greater deal of coordination of data (e.g., XML messages) to perform the task. Moreover, a branch that splits four ways is given more points than a three-way split which is given more points than a two-way split etc. Next, we move down a hierarchical level to all the services called by the top level and in turn consider what direct dependencies they have on their adjacent services. Thus, we move down the chain, level by level and the more services in the chain, the greater the overall score, and the greater the branching degree the higher the score. (The metric operates on the interval scale which means that one can meaningfully rank, add, average and subtract complexity scores but not consider ratios of metric values; the reader is referred to [13] for further details.)

In practice, it is very simple to compute the metric. First, one derives the tree of the complete service chain, complete meaning all leaves are durations. (Thus, one would need to substitute in and expand the WFS4 node, and any subservices it calls, in figure 2). Then, one assigns a score v_{seq} for each `seq` node and a score v_{sim} for each `sim` node and finally a score v_{leaf} for each terminal leaf. (While these leaves are durations, they are each assigned the same v_{leaf} score; the metric focuses solely on the structural, hierarchical complexity—the number of components and how they must be coordinated—rather than the absolute resources to complete each leaf.) Then, one simply sums these scores over the tree to obtain the final complexity score. In reality, v_{seq} and v_{sim} would represent a true cost function, say the computational cost (time, number of operations, file store required etc.) to coordinate the activity at that node. For simplicity in this generic model, we take $v_{seq} = v_{sim} = 3$ and $v_{leaf} = 1$ [c.f. 13]. For example, the complexity of the chain `seq(sim(`X,Y`),`Z`)` is $3 + 3 + 3 \times 1 = 9$.

Two important clarifications must be made here. First, the absolute and relative values for v_{seq}, v_{sim} and v_{leaf} (3,3,1) can be arbitrary as here, and in [13], yet it still provides a rigorous and objective final complexity score. (The objectivity is taken care of both from the rigorous set of rules by which computes the score and in the way in one can subsequently use these complexity values, e.g. to rank the complexity of different service chains.) However, as mentioned above, there is no reason that these scores cannot reflect true quantitative measures, such as CPU cycles or memory usage, of the difficulty of coordinating activity at `sim` and `seq` nodes.

The second clarification involves the grammar, specifically the number of arguments for `seq` and `sim` nodes. That is, an expression such as `seq(seq(`X,Y`),`Z`)` may appear two `seq` nodes but we could have easily have written `seq(`X,Y,Z`)` for the same *conceptual* meaning, and which has one `seq` node. Thus, while the simulation enforces a grammar where `seq` takes only two arguments, the complexity metric employed both here and in [13]

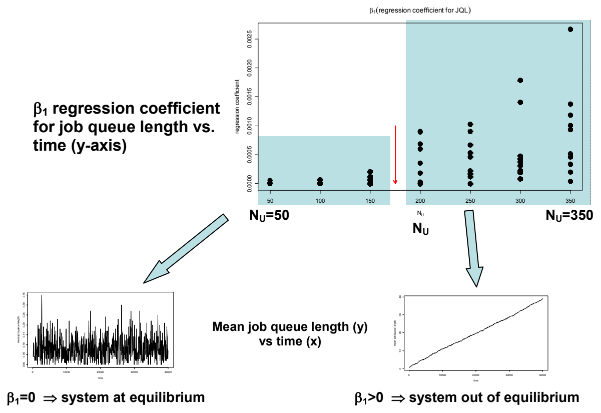


Figure 4: β_1 regression coefficient (gradient) for job queue length over time (y-axis) versus N_u (x-axis). N_s is constant at 100. Below N_u = 175 system dynamics are similar to that shown in the inset in bottom left while above N_u = 175, the system is clearly out of equilibrium with dynamics similar to that shown in the bottom right inset.

assumes a more general grammar with two or more arguments for `seq` nodes. In short, we calculate the complexity metric of the *most compact* representation possible collapsing nodes into parent nodes that are of the same type. Thus, `seq(seq(`X,Y`),` Z`)` is collapsed to the expression `seq(`X,Y,Z`)` and receives a complexity score of $3 + 1 + 1 + 1 = 6$.

Table 1: Standard Parameters Settings

Parameter	**Value**
Simulation-run-related	
Simulation duration (in aribtrary time units)	40000
Subservices and service chain-related	
Number of catalog subservices	1
Number of WCS subservices	40
Number of WFS subservices	40
Number of WMS subservices	40
Number of leaves in a subservice tree	U{2,3}
Probability that a server will host each service	0.25
Probability that a server hosts each of a service's subservices (given they host that service)	0.1
p_{leaf}: probability that a leaf will be a duration (versus a subservice call)	0.5
General parameters	
N_u: Number of users	500
N_s: Number of servers	500
$1 / \lambda$: Mean interdeparture interval of user requests (from each user)	500
Standard deviation of mean user request interdeparture interval	2
Mean number of subservices in a user's initial profile	2
Standard deviation of mean number of subervices in a user's initial profile	2
Probability that the user chooses a random subservice from a catalog (versus from its profile)	0.15
CPU polling type	FIFO
CPU update interval	5

2.6 Parameters and metrics

Table 1 lists our standard set of parameters, selected to produce an under-utilized system in equilibrium (such as those in the Figure 4 above). From this baseline, we systematically explored the effect of various parameters upon quality of service and system performance metrics. These user-related metrics include median (with respect to all users) number of outstanding requests per user (averaged over several time post-equilibrium time points per simulation) and the median service duration (i.e., the time to complete a typical request). For servers, they include median (with respect to all servers) number of jobs in a server's CPU job queue (averaged over several time points per simulation), the gradient of a regression of job queue length versus time for each server (a positive gradient implying server overloading), and total throughput (total number of requests completed by the suite of servers per simulation, which was of fixed duration). Finally, the following metrics were calculated on each full subservice chain: number of leaves, the minimum time in which a subservice chain could be completed if there were no delays (i.e., sum of leaf durations across the tree), subservice chain complexity metric (as defined earlier) and median efficiency (the theoretical minimum time to complete a particular service chain divided by the median duration it actually took to complete).

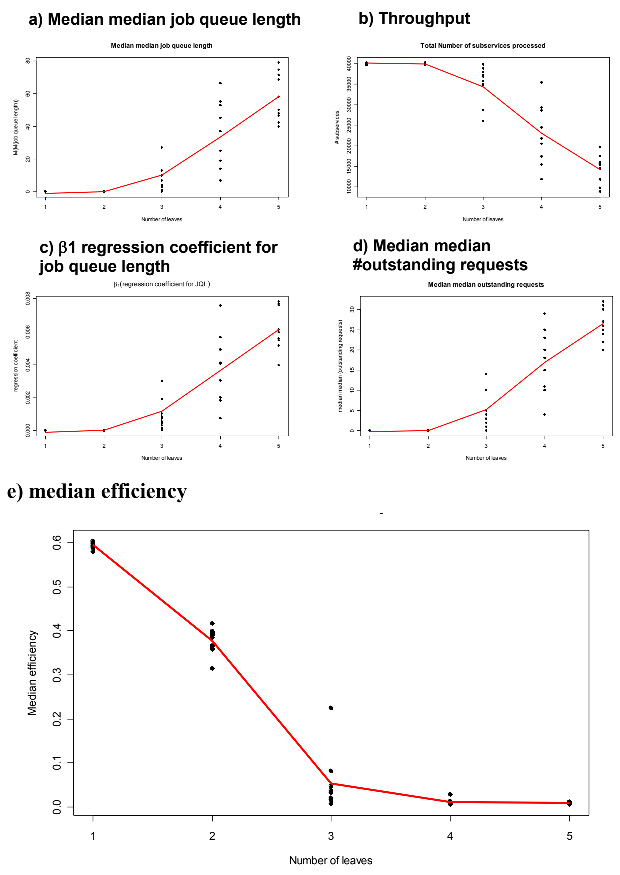


Figure 5: the effect of the number of leaves (from 1 – 5) in each subservice versus five quality of service and performance metrics. a) job queue length, b) total throughput, c) server loading (as measured by the gradient of a regression of job queue length versus time), d) number of outstanding requests for users and e) service chain efficiency. These data show that longer individual subservices lead to exponentially greater server loading, lowered throughput, efficiency, and end-user QoS.

3. Results

Numerous experiments were run to test whether basic quality of service metrics varied with system size (keeping the same user : server ratio). In runs with 10 – 10,000 users, the metrics were invariant (results not shown). As such, we are justified in using a relatively small number of users and servers (hence, leading to quicker simulations) as our baseline parameters.

Unfortunately, space limitation does not permit us to detail all of our results for all the parameters that we investigated.

3.1 Number of leaves in a subservice

Figure 5 shows the effect of increasing the number of leaves (from 1 – 5) in each single subservice versus four QoS and performance metrics. Job queue length (Figure 5a), regression coefficient (the proxy for server overloading, Figure 5c) and number of outstanding requests (Figure 5d) all increase exponentially with increasing number of leaves. The explanation is simple: more leaves in a single subservice tree lead to longer service chains, which means more processing time per job on servers leading to greater server utilization, and a lower turnover rate (Figure 5c) and hence longer time to complete each job and return it to the user. Consequently, service chain efficiency decreases with increasing number of leaves (Figure 6).

While this sequence is both expected and intuitive, the beauty and utility of this simulation tool is the ability to investigate objectively such qualitative relationships. These qualitative relationships should be relatively robust with respect to the particular parameter values uses, and moreover the QoS and performance metrics were deliberately chosen that these could be validated (in principle) from data, such as server logs, from real web services.

Other parameters such a p_{leaf} produce qualitatively similar results.

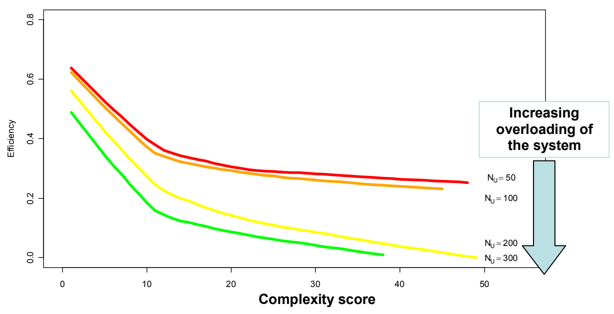
3.2 Service chain complexity

One of the most interesting results to come from this study, and one that would be very difficult to achieve without such a model, is the relationship between service chain complexity and efficiency. Figure 6a shows the relationships between these two metrics and number of users. N_s was fixed at 100 and N_u varied from 50 – 300. Each of the four curves, corresponding to a different number of users, is a LOWESS smoother fit [14] to the combined data from ten simulations (representing many hundreds of data points to which the curve is fitted).

There are several interesting results in these data. First, the higher the complexity score, the lower the efficiency; moreover, this relationship is roughly exponential, but perhaps more interestingly, there is distinct “elbow” in the curve at about complexity score 10. (As a guide, chains with two `sim` or `seq` nodes and four duration leaves, such as `seq`(`sim`(*a*,*b*,*c*),*d*) have a complexity score of ten.) Second, increasing network load decreases efficiency (the curves lower as N_u increases). Third, the more complex the chains are, the greater is this effect. For instance, the N_u = 200 curve (yellow) is a much lower proportion of the N_u = 50 (red) curve as complexity score increases.

Similarly, Figure 6b shows the median duration to complete a subservice request versus complexity score and N_u. Unsurprisingly, these show the opposite trends as figure 6a and perhaps show these in a more striking manner: the greater the service chain complexity score, the longer the duration, and the greater the system loading the far greater the effect.

a) service chain efficiency



b) subservice completion duration

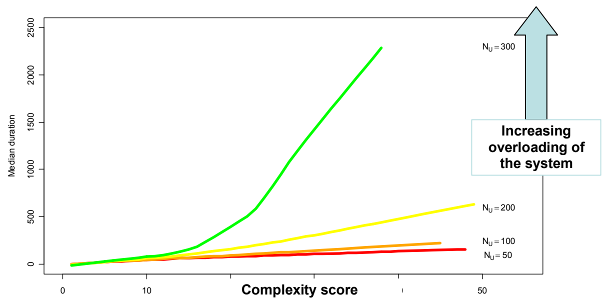


Figure 6: a) median service chain efficiency and b) median subservice competition duration versus service chain complexity for four values of number of users, N_u (Number of servers, N_s, is fixed at 100). Each curve is a smooth spline through several hundred data points. a) These data show that i) longer service chains are less efficient, and ii) that increasing network load makes service chains less efficient, and iii) the more complex they are, the greater the effect. b) These data show increasing network load makes service chains take longer to process, and the more complex they are, the greater the effect.

4. Discussion

In this work, we developed a number of tools and procedures for generic modeling and testing service chains at the network level. Not only did we demonstrate that one can orchestrate very complex service chains with hundreds of users and servers in a very simple and distributed manner (which is a very important, but easily overlooked result *per se* in this study: service chaining worked!) but our architecture allowed us to test some questions that can't currently be answered with current extant web services—e.g., what happens to end-user quality of service metrics as the average length of service chains increase. In short, we are essentially looking at issues that are several years down the road for actual web services. Ideally, if one can use a tool to pre-empt service chaining problems before the practitioners get to that development stage then this can be enormously useful and potentially be used to shape protocols, service API and even ISO specifications.

We developed a very simple grammar for defining web services. While this particular grammar may not be incredibly generic, the *approach*—the ability to define and generate random service chains using a genetic programming [15] type approach—is very useful, especially if dealing with a client with very specific, well-defined set of services. That is, this approach allows one to take a set of low-level, modular service building blocks and combine them in a multitude of valid combinations (i.e. that satisfy functional and API requirements) and test whether the protocols are correct. In short, this testing allows one to simulate, and again, *pre-empt*, all the service chains that users might construct from a given service suite.

The TouchGraph-based visualization provides a global view of network traffic (both from density of message traffic and size of bar charts representing job queue length at each server) getting across the scale of such a distributed system and the degree of coordination required.

Finally, we devised/adapted a metric that quantifies "service chain complexity" that takes into account both the number of elements and the dependency relations in a rigorous, objective, and quantitative manner. It is very generic and likely suitable for just about any current/future web service (chain), and this metric could be especially useful if one can make it more objective and assign the different component scores (e.g., the score for a team subtask) on the basis of real data: e.g. computational time/effort to achieve the subtask.

4.1 Future directions

While we strove for decentralized procedures wherever possible, the one component that was global was the "pecking order" algorithm that prevented cycles. In the real world, it would be possible for users to create these loops (inadvertently). In future research that would include distributed token-based loop checking algorithms and server timeout functionality, it would be interesting to allow these cycles and study their effects.

Also interesting would be to enhance the reality of user behavior. Currently, users submit a Poisson stream of completely independent requests. Of course, real users exhibit far more sophisticated behavior. First, users vary in their temporal behavior. For instance, some may exhibit a flurry of activity when they start work in the morning (e.g., obtaining the files they need to work on), before lunch (sending off completed work) and after lunch (obtaining new work) and before they leave in the evening (sending off new completed work). Such concentrated loads from multiple users can have a significant impact upon network dynamics. Second, users chose a new request type to submit entirely independent of previous request types submitted or received (excepting catalog request calls). In reality, users will often submit a request that depends greatly on the results of a request. Thus, a user monitoring a battlefield who submits a request for a detailed satellite image of a possible enemy sighting, will submit different additional requests depending upon whether the image confirms that the sighting is friend or foe. Interestingly, user request sequences can be modeled in exactly the same way as service chains themselves using the service chain grammar described earlier.

Finally, our network is idealized in that we implicitly assume infinite bandwidth, and that servers are always operational. Both of these features can dramatically affect network dynamics. For instance, the basis of web services is that they can form *ad hoc* networks: if one server that hosts a necessary service is down then one can always use a catalog service to find another host. However, if many users or servers do this simultaneously, this can lead to network instability and cascading effects as traffic switches to one overloaded server to the next.

The crux this future work is to ground the simulation model closer to reality and introduce some additional features that will provide further insights into web service network dynamics and quality of service for the end users, potential problems that may arise and potential solution architectures and algorithms, focusing on those solutions that are distributed, scalable and robust.

Acknowledgements

We thank the Defense Information Systems Agency (DISA) for funding this research (contract DCA-200-00-5018). We also thank Charles Heazel of the Open Geospatial Consortium (OGC) for his help, support and insights throughout the work.

References

[1] Bloomberg, J. Service Orientation Market Trends Report. Predicting the Future of XML & Web Services. Unpublished report. http://www.zapthink.com/report.html?id=ZTR-WS110. 2004 (last accessed February 8, 2005).

[2] ISO/TC-211. Geographic Information – Services. ISO/DIS 19119. 2002

[3] Percivall, G. ISO 19119 and OGC Service architecture. In: *FIG XXII International Congress, Washington DC, USA, April 19-26, 2002*.

[4] Alameh, N. Chaining Geographic Information Web Services. *IEEE Internet Computing*, 2003, **7**(5), 22–29.

[5] Alameh, N. Service chaining of interoperable geographic information web services, 2002. http://web.mit.edu/nadinesa/www/paper2.pdf. (Accessed February 7, 2005).

[6] Aditya, T., and Lemmens, R.L.G. Chaining distributed GIS services. In: Prosiding Pertemuan Ilmiah Tahunan XII, Masyarakat Penginderaan Jauh Indonesia : Inovasi dan modifikasi teknologi penginderaan jauh dan SIG untuk pengembangan program Kelatuan dan Pertanian di Indonesia, Bandung, 29-30 Juli 2003. 8 p.

[7] McShea, D.W. Metazoan complexity and evolution: is there a trend? *Evolution*, 1996, **50**(2), 477–492.

[8] Bonabeau, E. Agent-based modeling: Methods and techniques for simulating human systems. *Proceedings of the National Academy of Sciences USA*, 2002, **99**, 7280–7287.

[9] Einspanier, U., M. Lutz, K. Senkler, I. Simonis, and A. Sliwinski. Toward a Process Model for GI Service Composition, GI-Tage (GI Days), Institute for Geoinformatics, University of Münster, 2003. http://ifgi.uni-muenster.de/~simonis/download/gitage2003b.pdf (accessed February 7, 2005).

[10] http://www.opengeospatial.org/

[11] http://ip.opengis.org/ows2/

[12] http://www.touchgraph.com/

[13] Anderson, C., N.R. Franks, and D.W. McShea. The complexity and hierarchical nature of tasks in insect societies. *Animal Behaviour*, 2001, **62**, 643–651.

[14] Cleveland, W.S. LOWESS: A program for smoothing scatterplots by robust locally weighted regression. *The American Statistician*, 1981, **35**, 54–54.

[15] Koza, J. *Genetic Programing: on the Programming of Computers by Means of Natural Selection*, MIT Press, Cambridge, Mass., 1992.