Replicating Web Services for Scalability

Mario Bravetti[1], Stephen Gilmore[2], Claudio Guidi[1], and Mirco Tribastone[2]

[1] University of Bologna
[2] University of Edinburgh

Abstract. Web service instances are often replicated to allow service provision to scale to support larger population sizes of users. However, such systems are difficult to analyse because the scale and complexity inherent in the system itself poses challenges for accurate qualitative or quantitative modelling. We use two process calculi cooperatively in the analysis of an example Web service replicated across many servers. The SOCK calculus is used to model service-oriented aspects closely and the PEPA calculus is used to analyse the performance of the system under increasing load.

1 Introduction

Web Services expose applications on the Internet for open, accessible use. The computational dynamics of such a distribution are that the resources of a server hosting a service endpoint are shared among its many, geographically distributed, clients. Evidently such a single-server design cannot scale to accommodate very large numbers of clients so when scalability is identified as a concern a crucial enhancement to this deployment architecture is to replicate the service across many, usually geographically distributed, servers. This deployment leads to a scalable design where more clients can be accommodated by adding more servers. The resources of the replicated services are federated to serve many clients.

Clients of such a distributed service will usually need to become more complicated because they will first need to discover service endpoints before binding to a particular service instance. Service providers must also register with a registry of services, so that they may later be discovered by the clients. Some services are sufficiently specialised that their locations are known and this knowledge is built into the service composition, and exploited. We consider such a scenario here.

Web Services provide all of the necessary infrastructure for services to be deployed in this way, with formal statements of the service provided, a formal notion of registration with the registry and a procedure for service discovery in registries. In the present paper we are concerned with the analysis of the high-level design of a replicated service, based on measurements of individual service instances and probabilistic estimates of likely bindings chosen by clients.

We are concerned here with using process calculus models to investigate how well a distributed system can balance load in order to provide a scalable service for larger pools of users downloading content over a shared network. The specific scenario which we consider as an illustration of this class of systems is a Distributed E-Learning and Course Management System (DCMS) which provides management of courses and degrees offered at several co-operating universities, implemented as a collection of services. The system encompases services to provide e-learning courses which can be shared between universities and services which enable several universities to jointly provide e-learning courses, thus federating resources and providing a wider programme of courses of study than would be found at any of the universities individually. Lightweight federation of resources in this way to form a "virtual university" is exactly the type of interaction envisaged by the architects of the Web Service vision.

One of the difficulties of modelling such a design is capturing behaviour correctly, and assuring oneself and others that this has been achieved. We model the behaviour of the system in the SOCK calculus [1, 2]. We have exercised this model using the JOLIE interpreter [3] in order to increase our confidence that the model describes the behaviour which we intended to capture.

Another challenge of this type of work is the well-known state-space explosion problem whereby a formal model of a system to which the algebraic methods and tools of concurrency theory can be applied would be very likely to be resistant to effective formal analysis. State-space explosion arises because the size of the system as a whole is bounded by the product of the individual state-spaces of the components which are composed in parallel. Evidently, this grows very quickly even when the components used are high-level abstact models of services which incorporate only the essential details needed for the modelling study. Due to the state-space explosion problem models might be require either an infeasibly long time to analyse, or an infeasibly large amount of storage.

To address this challenge, and be able to model the scalability problem of interest, we adopt a continuous-space representation of the process algebra model in contrast to the usual discrete-state representation of process algebra models via labelled transition systems. The process algebra used, PEPA, and the continuous-state representation are both due to Hillston [4, 5]. The continuous-state representation avoids the requirement to represent each possible state of the system, making this analysis method applicable to systems of vastly greater scale and complexity than those analysable using the explicit, discrete-state representations which are usually based on Continuous-Time Markov Chains (CTMCs). In contrast the continuous-state representation maps the process algebra model to a system of coupled Ordinary Differential Equations (ODEs). Because of this an entirely different arsenal of numerical analysis procedures are available which can efficiently compute valuable analysis results for large-scale systems such as the one considered here.

Structure of this paper In Section 2 we describe related work. In Section 3 we present the Service Oriented Computing Kernel (SOCK) calculus used in Section 4 to model our example Web Service. Following this we introduce Performance Evaluation Process Algebra (PEPA) in Section 5 which we use to analyse the non-functional aspects of the example in Section 6. We conclude in Section 7.

2 Related work

There are now many papers where stochastic process calculus models are mapped to Continuous-Time Markov Chains for performance analysis [6–8]. Hillston's method of mapping process calculus models to ordinary differential equations is a more recent development [5] but has already been used to analyse peer-to-peer systems [9] and internet-scale spread of computer viruses such as worms [10]. An earlier paper by two of the present authors used Hillston's ODE method to show the failure of a centralised server model for the DCMS e-learning system to scale with increasing load [11].

3 The SOCK Calculus

SOCK (Service Oriented Computing Kernel) [1] is a formal calculus developed for reasoning about the main Service Oriented Computing issues. SOCK is divided into three different calculi which addresses different aspects of service design. The three SOCK calculi are called: *service behaviour calculus*, *service engine calculus* and *services system calculus*. The first one allows for the design of service behaviours by supplying computation and external communication primitives inspired by Web Services operations and workflow operators (e.g. sequence, parallel and choice). The service engine calculus is built on top of the former and allows for the specification of the service declaration where it is possible to design in an orthogonal way three main features: *execution modality*, *persistent state* flag and *correlation sets*. The execution modality deals with the possibility of executing a service in a sequential order or in a concurrent one; the persistent state flag allows the designer to declare if each session (of the service engine) has its own independent state or if the state is shared among all the sessions of the same service engine; correlation sets is a mechanism for distinguishing sessions initiated by different invokers by means of the values received within some specified variables. Finally, the services system calculus allows for the composition of service engines into a system.

The term syntax of the calculus includes numerical values and (possibly empty) tuples of variables $\boldsymbol{x} = \langle x_0, x_1, \ldots, x_n \rangle$ and values $\boldsymbol{v} = \langle v_0, v_1, \ldots, v_n \rangle$. The null process is $\mathbf{0}$. Operations are single message ($\mathcal{O}$) or involve two messages ($\mathcal{O}_r$). Outputs can be a signal s, a notification $o@k(\boldsymbol{x})$ or a solicit-response $o_r@k(\boldsymbol{x}, \boldsymbol{y})$ where o is an operation in $\mathcal{O}$, o_r in $\mathcal{O}_r$, k the receiver location, $\boldsymbol{x}$ the tuple of variables sent and $\boldsymbol{y}$ the received information. The process term $x := e$ denotes an assignment. $\chi?P : Q$ is the if-then-else process. $P;P$ is sequential composition and $P \mid P$ is parallel. Guarded choice is $P + P$. $\chi \leftrightharpoons P$ is guarded iteration. For a complete description the reader is referred to [1].

A brief discussion of the SOCK operators for service engine description and execution is given below:

Persistence The flags $\times$ and $\bullet$ are used to distinguish persistent and non-persistent state. Where P is a service behaviour then $P_\times$ is equipped with a non-persistent state whereas $P_\bullet$ is equipped with a persistent state.

Guards The execution of sessions may be guarded by correlation sets. In the term $c \triangleright P_\bullet$ the correlation set c guards the execution of the persistent service P. Correlation sets may be empty ($\emptyset$).

Sessions $!W$ denotes a concurrent execution of the sessions in W whereas W^* denotes that sessions are executed in sequential order. For example $!(\emptyset \triangleright P_\bullet)$ indicates the concurrent execution of uncorrelated persistent service P.

Engines A service engine Y is the composition of a service declaration D and an execution environment H, denoted $D[H]$. H represents the actual sessions which are running on the engine coupled with a state $(P, \mathcal{S})$.

Locations A service engine system E can be a located service engine Y_{LOC} or a parallel composition of them $E \parallel E$.

4 Modelling Behaviour with SOCK

In an e-learning system the teaching material prepared by the teaching staff of each university is made available as *learning objects* which students must obtain by download from the content servers of the universities involved. The learning objects contain electronic versions of course notes and presentation material such as lecture slides. In addition many learning objects contain digital audio or digital video recordings of lecture presentations given by teaching staff. Learning objects are compressed archives of teaching material which vary in size and scale from collections of material for a single lecture in a course to a complete record of an entire lecture course. The lecture presentations of the course are downloaded instead of being streamed because they may require repeated review in order to digest the content.

Universities which host e-learning content are concerned with providing services which ensure good availability of the content and limited download times for the learning objects. Both of these are considered to be important metrics and are addressed in different ways. A high level of availability of the content is ensured by replicating the content distribution services (and the associated learning objects) across the content servers of many of the universities involved. Download times are reduced where possible by binding content requestors at the point of download to the content server which is most likely to be able to serve them well at that time.

The dynamic choice of content server is made using a metric which takes into account the geographical location of the content requestor and the content server, available bandwidth between the hosts, and the current load on the content server. Some of these factors can be known or bounded in advance (e.g. the maximum possible bandwidth between two endpoints) but some values must be obtained at the time that the service is invoked (e.g. the current load on a server).

It might seem that the best choice of server should always be the one which is geographically closest however it is possible that a lightly-loaded server further away from the content requestor might be able to serve them more quickly than a heavily-loaded server which is nearby. When considering home download it

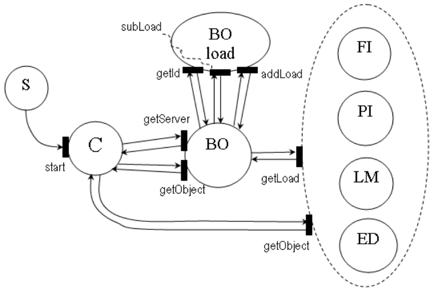


Fig. 1. The configuration of servers and services at the five sites

is usually the bandwidth to the Internet Service Provider which is the limiting factor on download rate in any case. The metric used by the dynamic discovery service attempts to take location, bandwidth and load factors into account in order to be able to make a good selection of content host for the content requestor.

Below we describe in the SOCK calculus the policy which would be used at the Bologna site to determine the selection of content server. Each of the content servers provides a service `getLoad` which, when invoked returns the current load on the server as a integer value in the range 0 (no load, available for use) to 100 (fully loaded, unavailable for use). Lower numbers are better. The policy at the Bologna site (`UNIBO`) compares its own load with the load at Pisa (`UPISA`), Florence (`UNIFI`), Munich (`LMU`) and Edinburgh (`UEDIN`) before returning the name of the server to download from. The remote servers are checked in a priority order, with geographically nearer servers being checked before those which are further away. A graphical representation of the system is shown in Figure 1.

4.1 Model in SOCK

In this section we present the SOCK behaviour of the services involved in the system, together with their deployment in terms of SOCK service engines concurrently composed within the process *System*. The names *UNIBO*, *UPISA*, *UNIFI*, *LMU* and *UEDIN* abstractly represent the location of the services provided by the universities of Bologna, Pisa, Firenze, Munich and Edinburgh, respectively. In particular, three behaviours are described: the *clientBehaviour*, the *UniBoBehaviour* and the *ObjServerBehaviour*.

The *clientBehaviour* models the behaviour of a client which sends a request to the service of the University of Bologna by exploiting the Solicit-Response *getServer@UNIBO* and, as a reply, it receives the address of the service to invoke for retrieving the e-learning object it is looking for. If the response message contains a valid address (here we model a fault reply message with the value -1), the client downloads the e-learning object by invoking the *getObject* operation of the service whose location has been stored within the variable *ServerAddress*. Here, we exploit the value *id_value*, assigned to the variable *ObjectID*, for modelling the reference of the object to download and we suppose that all the servers are able to provide the same e-learning objects.

$$\begin{aligned} \textit{clientBehaviour} ::= \; & \textit{getServer}@\textit{UNIBO}(\langle\rangle, \textit{ServerAddress}); \\ & \textit{ServerAddress} == -1?\mathbf{0} : \\ & \quad (\textit{objectID} := \textit{id\_value} \\ & \quad ; \textit{getObject}@\textit{ServerAddress}(\textit{objectID}, \textit{object})) \end{aligned}$$

The *UniBoBehaviour* models the behaviour of the service provided by the University of Bologna and it supplies two different operations: *getServer* and *getObject*.

$$\begin{aligned} \textit{UniBoBehaviour} ::= \; & \textit{getServer}(\langle\rangle, \textit{addr}, \textit{search}) \\ & +\textit{getObject}(\textit{id}, \textit{obj}, \textit{obj} := \texttt{retrieve\_obj(}\textit{id}\texttt{)}) \end{aligned}$$

The former allows for the retrievement of the downloading service by following a policy that takes into account the load of each server, whereas the latter allows for the downloading of an e-learning object directly from the UNIBO service. It is worth noting that the load of the other servers is retrieved by exploiting the Solicit-Response operation *getLoad* whereas the functions `loadhere()` and `retrieve_obj()` model the internal computations for calculating the actual load of the UniBo server and retrieving the requested object from the internal database of the server, respectively.

$$\begin{aligned} \textit{search} ::= \; & \textit{load} := \texttt{loadhere()}; \textit{load} < 75?\textit{addr} := \textit{UNIBO} \\ & \quad : \textit{getLoad}@\textit{UNIFI}(\langle\rangle, \textit{load}); \textit{load} < 60?\textit{addr} := \textit{UNIFI} \\ & \quad\;\; : \textit{getLoad}@\textit{UPISA}(\langle\rangle, \textit{load}); \textit{load} < 60?\textit{addr} := \textit{UPISA} \\ & \quad\quad : \textit{getLoad}@\textit{LMU}(\langle\rangle, \textit{load}); \textit{load} < 40?\textit{addr} := \textit{LMU} \\ & \quad\quad\;\; : \textit{getLoad}@\textit{UEDIN}(\langle\rangle, \textit{load}); \textit{load} < 20?\textit{addr} := \textit{UEDIN} \\ & \quad\quad\quad : \textit{load} := \texttt{loadhere()}; \textit{load} < 95?\textit{addr} := \textit{UNIBO} \\ & \quad\quad\quad\;\; : \textit{addr} := -1 \end{aligned}$$

Finally, the *ObjServerBehaviour* models the behaviour of each downloading service by providing two different Request-Response operations: *getLoad* and *getObject*. The former allows for the returning of the load of the server whereas the latter provides a means for downloading the requested e-learning object.

$$\begin{aligned} \textit{ObjServerBehaviour} ::= \; & \textit{getLoad}(\langle\rangle, \textit{load}, \textit{load} := \texttt{loadhere()}) \\ & +\textit{getObject}(\textit{id}, \textit{obj}, \textit{obj} := \texttt{retrieve\_obj(}\textit{id}\texttt{)}) \end{aligned}$$

As far as the deployment of the services is concerned, below six service engines are composed within a process called *System*. For the sake of precision, the *client* is not a service because it does not start with a receiving operation thus, its service engine provides only an execution environment, without any declaration, where the service behaviour can be executed once. The *UniBoServer* is the service engine which executes the *UniBoBehaviour* whereas *UPisaServer*, *UniFiServer*, *LmuServer* and *UEdinServer* are the service engines of the downloading servers which all execute the same behaviour *ObjServerBehaviour* but at different locations.

$$
\begin{aligned}
\mathit{client} &::= [\mathit{clientBehaviour}]_{\mathit{CLIENT}} \\
\mathit{UniBoServer} &::= !(\emptyset \triangleright \mathit{UniBoBehaviour}_{\bullet})[\emptyset \triangleright (\mathbf{0}, \mathcal{S})]_{\mathit{UNIBO}} \\
\mathit{UniFiServer} &::= !(\emptyset \triangleright \mathit{ObjServerBehaviour}_{\bullet})[\emptyset \triangleright (\mathbf{0}, \mathcal{S})]_{\mathit{UNIFI}} \\
\mathit{UPisaServer} &::= !(\emptyset \triangleright \mathit{ObjServerBehaviour}_{\bullet})[\emptyset \triangleright (\mathbf{0}, \mathcal{S})]_{\mathit{UPISA}} \\
\mathit{LmuServer} &::= !(\emptyset \triangleright \mathit{ObjServerBehaviour}_{\bullet})[\emptyset \triangleright (\mathbf{0}, \mathcal{S})]_{\mathit{LMU}} \\
\mathit{UEdinServer} &::= !(\emptyset \triangleright \mathit{ObjServerBehaviour}_{\bullet})[\emptyset \triangleright (\mathbf{0}, \mathcal{S})]_{\mathit{UEDIN}} \\
\mathit{System} &::= \mathit{client} \parallel \mathit{UniBoServer} \parallel \mathit{UPisaServer} \parallel \mathit{UniFiServer} \\
&\quad\; \parallel \mathit{LmuServer} \parallel \mathit{UEdinServer}
\end{aligned}
$$

5 The PEPA Stochastic Process Algebra

Systems are represented in PEPA as the composition of *components* which undertake *actions*. In PEPA the actions are assumed to have a duration, or delay. Thus the expression $(\alpha, r).P$ denotes a component which can undertake an α action, at rate r to evolve into a component P. Here $\alpha \in \mathcal{A}$ where $\mathcal{A}$ is the set of action types and $P \in \mathcal{C}$ where $\mathcal{C}$ is the set of component types. The rate r models a delay of variable duration. Delays are samples from an exponential random variable with parameter r, where this parameter is most often constant. In this paper we will make use of *functional rates* [12] which allow the rate at which an activity is performed to depend on the current state of the model. (In Petri nets terms, a "marking-dependent" rate.)

For example, a server might offer its computing resources at a rate which depended on the current state, $(\mathit{compute}, f_{SERVER})$ where the function f_{SERVER} is defined as follows:

$$
f_{SERVER} = \begin{cases} 0, \text{ if } \mathit{Server}_{down} \\ \lambda, \text{ if } \mathit{Server}_{up} \end{cases}
$$

A full description of the PEPA language can be found in [4]. To briefly summarise, PEPA has a small set of combinators, prefix (.), choice (+), co-operation ($\bowtie$, when co-operating over a set of activities, or $\parallel$ when there is no co-operation) and hiding (which we will not use here). Because we will be working with large populations of replicated processes we write $P[n]$ to denote n copies of component P executing in parallel. For example,

$$
P[5] \equiv (P \parallel P \parallel P \parallel P \parallel P).
$$

The total capacity of a component P to carry out activities of type α is termed the *apparent rate* of α in P, denoted $r_\alpha(P)$. For example, $r_{compute}(\mathit{Server}_{up}[2]) = 2\lambda$, $r_{compute}(\mathit{Server}_{up} \parallel \mathit{Server}_{down}) = \lambda$, and $r_{compute}(\mathit{Server}_{down}[2]) = 0$.

5.1 Relating Markov chains and ODEs

In performance modelling based on continuous-time Markov chains, measures of system performance are often derived by a calculation which uses the steady-state probability distribution. To help us to compare modelling with ODEs and CTMCs in this section we consider the simpler example of a queue in PEPA.

$$\begin{aligned} Q_0 &\stackrel{\mathit{def}}{=} (\mathit{arrive}, \lambda).Q_1 \\ Q_i &\stackrel{\mathit{def}}{=} (\mathit{arrive}, \lambda).Q_{i+1} + (\mathit{serve}, \mu).Q_{i-1} \qquad (0 < i < N) \\ Q_N &\stackrel{\mathit{def}}{=} (\mathit{serve}, \mu).Q_{N-1} \end{aligned}$$

A typical performance measure for a model based on queues is the average queue length, which is computed in different ways, depending on the observations offered by the chosen semantics for the interpretation of the model.

When modelling in the Markovian interpretation we obtain the steady-state probability distribution, $\boldsymbol{\pi}$. For a given queue bound, say $N = 8$, the average queue length is computed by weighting the probability of a state (Q_i denotes the state where the queue is of length i) by the number of customers in the queue at that point.

$$a = \sum_{i=0}^{8} i\pi(i)$$

When the state-space of the model grows in size any analysis which is based on an interleaving semantics (as in CTMCs) becomes prohibitively expensive. We turn then to a continuous approximation and solve the *initial value problem* for the ODEs to see how the numbers of each type of component change from initial (known) values at time $t = 0$, as time progresses forwards. We cannot compute the average queue length in the same way as for the CTMC because we do not have the stationary probability distribution. Instead we calculate it by considering a collection of 90 (say) independent queues all of capacity 8. The average queue length at time t is

$$a = \sum_{i=0}^{8} i \frac{[Q_i(t)]}{90}$$

where the term $[Q_i(t)]$ is understood to mean "the number of instances of Q_i at time t". We divide by 90 because that is the number which we have in our collection in this example.

We compute the average queue length numerically using both CTMC-based and ODE-based approaches, up to a specified accuracy of the numerical solution procedures (that is, a specified number of decimal places of accuracy). When we compare these we find good agreement in the results, up to the specified accuracy of the calculation of the solutions (see Figure 2). The solutions are computed using two entirely different numerical procedures. For the Markov chain, Jacobian over-relaxation, and for the differential equations, fifth-order Runge-Kutta with an adaptive step size.

		Av. queue length	Av. queue length	Difference
λ	μ	(CTMCs at equilibrium)	(ODEs at $t = 200$)	
1	4	0.333299009029	0.333298753978	2.5×10^{-7}
1	2	0.982387959648	0.982386995556	9.6×10^{-7}
1	1	4.000000000000	4.000000266670	-2.6×10^{-7}
2	1	7.017612040350	7.017613704440	-1.6×10^{-6}
4	1	7.666700990970	7.666701306580	-3.2×10^{-7}

Fig. 2. Solutions computed using CTMCs and ODEs

It is pleasing to have such good agreement in the results but it might be something of a mystery to the reader as to why the agreement is so good. In order to illuminate further the relationship between the CTMC and ODE interpretations we consider a simpler instance of the model above, a single sequential component with only three states defining a two-place queue.

$$\begin{aligned}
Q_0 &\stackrel{\mathit{def}}{=} (\mathit{arrive}, \lambda).Q_1 \\
Q_1 &\stackrel{\mathit{def}}{=} (\mathit{arrive}, \lambda).Q_2 + (\mathit{serve}, \mu).Q_0 \\
Q_2 &\stackrel{\mathit{def}}{=} (\mathit{serve}, \mu).Q_1
\end{aligned}$$

The continuous-time view This process is at least enough to contain a use of a choice (in Q_1). When interpreted against the operational semantics of Markovian PEPA [4] this generates the following generator matrix for the underlying Markov chain. (By convention this matrix is called $\mathbf{Q}$, but it is not to be confused with our process variables Q_0, Q_1 and Q_2).

$$\mathbf{Q} = \begin{bmatrix} -\lambda & \lambda & 0 \\ \mu & -\lambda - \mu & \lambda \\ 0 & \mu & -\mu \end{bmatrix}$$

The stationary probability distribution of this Markov chain, $\boldsymbol{\pi}$, is obtained by solving the equation

$$\boldsymbol{\pi}\mathbf{Q} = \mathbf{0}$$

subject to the requirement that the distribution is a good probability distribution (i.e. sums to 1).

$$\sum \boldsymbol{\pi} = 1$$

The symbolic solution of the above set of simultaneous linear equations is

$$\boldsymbol{\pi} = \left[\frac{\mu^2}{\lambda^2 + \mu\,\lambda + \mu^2}, \frac{\mu\,\lambda}{\lambda^2 + \mu\,\lambda + \mu^2}, \frac{\lambda^2}{\lambda^2 + \mu\,\lambda + \mu^2} \right].$$

The continuous-space view When interpreted against the ODE semantics of PEPA [5], the above model instead gives rise to the following system of ordinary differential equations.

$$\begin{aligned}
\frac{dQ_0}{dt} &= -\lambda Q_0 + \mu Q_1 \\
\frac{dQ_1}{dt} &= \lambda Q_0 - \lambda Q_1 - \mu Q_1 + \mu Q_2 \\
\frac{dQ_2}{dt} &= \lambda Q_1 - \mu Q_2
\end{aligned}$$

A system of differential equations has a stationary solution, which occurs, as you might expect, when nothing is changing. That is, for our queue:

$$\begin{aligned}
0 &= -\lambda Q_0 + \mu Q_1 \\
0 &= \lambda Q_0 - \lambda Q_1 - \mu Q_1 + \mu Q_2 \\
0 &= \lambda Q_1 - \mu Q_2
\end{aligned}$$

If we re-write the above system of linear equations in vector-matrix form, we find that it is:

$$\mathbf{0} = [Q_0 \quad Q_1 \quad Q_2]\mathbf{Q}$$

If we then solve this initial value problem for the above system of differential equations for initial values of $Q_0 = 1, Q_1 = 0, Q_2 = 0$ then, because of conservation of mass, the equilibrium points will coincide with the steady-state distribution of the CTMC model. Therefore all measures calculated from the steady-state probability distribution (such as average queue length) will coincide. We argued this agreement only by considering one simple example here but a formal correspondence between the two semantic descriptions has been proven by Hillston by reference to Kurtz's theorem.

6 Modelling Performance with PEPA

The distributed system in PEPA is based on the cooperation between a population of clients and instances of server threads at each mirror site. Let m be the number of classes of clients in the system and k the number of mirror sites. In this modelling framework, the distributed system is completely characterised by the following entities:

- *Connection Setup Matrix* $\mathbf{C} \in {\mathbb{R}^+}^{m,k}$, whose element $c_{i,j}$ is the rate at which a class-i client connects to mirror j.
- *End-to-End Available Bandwidth Matrix* $\mathbf{D} \in {\mathbb{R}^+}^{m,k}$, whose element $d_{i,j}$ is the rate at which a class-i client downloads from mirror j.
- *Idle Vector* $\mathbf{t} \in {\mathbb{R}^+}^{m}$, whose element $r_{idle,i}$ is a class-i client's *thinking time.*
- *Population Vector* $\mathbf{p} \in {\mathbb{N}^+}^{m}$, whose element p_i is the population of class-i clients.

- *System Deployment Vector* $\mathbf{q} \in {\mathbb{N}^{+}}^{k}$, whose element q_j denotes the number of threads available at mirror j.

The model of a client is as follows.

$$\begin{aligned}
\mathit{Client}_i &\overset{\mathit{def}}{=} (\mathit{connect}_1, c_{i,1}).(\mathit{download}_1, d_{i,1}).\mathit{Idle}_i\\
&+ (\mathit{connect}_2, c_{i,2}).(\mathit{download}_2, d_{i,2}).\mathit{Idle}_i\\
&\dots\\
&+ (\mathit{connect}_k, c_{i,k}).(\mathit{download}_k, d_{i,k}).\mathit{Idle}_i\\
&+ (\mathit{overload}, \top).\mathit{Client}_i\\
\mathit{Idle}_i &\overset{\mathit{def}}{=} (\mathit{idle}, r_{\mathit{idle},i}).\mathit{Client}_i
\end{aligned}$$

$$(1 \leq i \leq m)$$

Although the clients attempt connections to all the mirrors, we will model the mirrors in such a way that only one connection is granted as determined by the policy expressed below. For each mirror $Mirror_j, 1 \leq j \leq m$, we have:

$$\begin{aligned}
\mathit{Mirror}_j &\overset{\mathit{def}}{=} \big(\mathit{connect}_j, f_j(s)\big).\mathit{MirrorUploading}_j\\
\mathit{MirrorUploading}_j &\overset{\mathit{def}}{=} \big(\mathit{download}_j, \top\big).\mathit{Mirror}_j
\end{aligned}$$

This description features a functional rate for the *connect* action, $f_j(s) : \mathcal{C} \rightarrow \{0, \top\}$ where s is a PEPA component denoting the current state of the system. When f_j evaluates to 0, the activity is not enabled by the sequential component. We have determined that in any state at most one such function evaluates to $\top$, i.e.:

$$\forall s, \nexists f_i, f_j : f_i(s) = \top, f_j(s) = \top, j \neq i$$

By defining the functional rates for the *connect* action we encode the load balancer's policy into the model, as we shall see later. Note that no mirror performs any *overload* action. This is accomplished by another sequential component as follows:

$$\mathit{Overload} \overset{\mathit{def}}{=} \big(\mathit{overload}, o(s)\big).\mathit{Overload}$$

$$o(s) = \begin{cases} \top & f_i(s) = 0, 1 \leq i \leq m\\ 0 & \text{otherwise} \end{cases}$$

That is, *Overload* is enabled if all the mirrors' functional rates evaluate to 0. This ensures that no state is deadlocked. The initial state of the system is:

$$\begin{aligned}
&\big(\mathit{Client}_1[p_1] \parallel \mathit{Client}_2[p_2] \parallel \dots \parallel \mathit{Client}_m[p_m]\big)\\
&\qquad \underset{L}{\bowtie} \big(\mathit{Mirror}_1[q_1] \parallel \mathit{Mirror}_2[q_2] \parallel \dots \parallel \mathit{Mirror}_k[q_k]\big)
\end{aligned}$$

$$\begin{aligned}
L = \{&\mathit{connect}_1, \mathit{connect}_2, \dots \mathit{connect}_k,\\
&\mathit{download}_1, \mathit{download}_2, \dots \mathit{download}_k, \mathit{overload}\}
\end{aligned}$$

Let *Loc* be a k-tuple assigning labels to mirrors, so that we can use Mirror_j and $\mathit{Mirror}_{\mathit{Loc}_j}$ interchangeably. We now provide the model using the framework described above. In this case study, let $Loc = (\text{UNIBO, UNIFI, UPISA, LMU, UEDIN})$. In this example, we consider a single class of clients located at UNIBO, i.e. $m = 1$. In the definitions of the functional rates f_{UNIBO}–f_{UEDIN}, we use process terms to indicate the number of copies of sequential components that behave as those terms in the system's state. The functional rates are defined thus.

$$f_{\text{UNIBO}} = \begin{cases} \top & \text{if } \mathit{MirrorUploading}_{\text{UNIBO}} < 75 \\ \top & \text{if } \mathit{MirrorUploading}_{\text{UNIBO}} < 95, \\ & \quad \mathit{MirrorUploading}_{\text{UNIFI}} \geq 60, \\ & \quad \mathit{MirrorUploading}_{\text{UPISA}} \geq 60, \\ & \quad \mathit{MirrorUploading}_{\text{LMU}} \geq 40, \\ & \quad \mathit{MirrorUploading}_{\text{UEDIN}} \geq 20 \\ 0 & \text{otherwise} \end{cases}$$

$$f_{\text{UNIFI}} = \begin{cases} \top & \text{if } \mathit{MirrorUploading}_{\text{UNIBO}} \geq 75, \\ & \quad \mathit{MirrorUploading}_{\text{UNIFI}} < 60 \\ 0 & \text{otherwise} \end{cases}$$

$$f_{\text{UPISA}} = \begin{cases} \top & \text{if } \mathit{MirrorUploading}_{\text{UNIBO}} \geq 75, \\ & \quad \mathit{MirrorUploading}_{\text{UNIFI}} \geq 60, \\ & \quad \mathit{MirrorUploading}_{\text{UPISA}} < 60 \\ 0 & \text{otherwise} \end{cases}$$

$$f_{\text{LMU}} = \begin{cases} \top & \text{if } \mathit{MirrorUploading}_{\text{UNIBO}} \geq 75, \\ & \quad \mathit{MirrorUploading}_{\text{UNIFI}} \geq 60, \\ & \quad \mathit{MirrorUploading}_{\text{UPISA}} \geq 60, \\ & \quad \mathit{MirrorUploading}_{\text{LMU}} < 40 \\ 0 & \text{otherwise} \end{cases}$$

$$f_{\text{UEDIN}} = \begin{cases} \top & \text{if } \mathit{MirrorUploading}_{\text{UNIBO}} \geq 75, \\ & \quad \mathit{MirrorUploading}_{\text{UNIFI}} \geq 60, \\ & \quad \mathit{MirrorUploading}_{\text{UPISA}} \geq 60, \\ & \quad \mathit{MirrorUploading}_{\text{LMU}} \geq 40, \\ & \quad \mathit{MirrorUploading}_{\text{UEDIN}} < 20 \\ 0 & \text{otherwise} \end{cases}$$

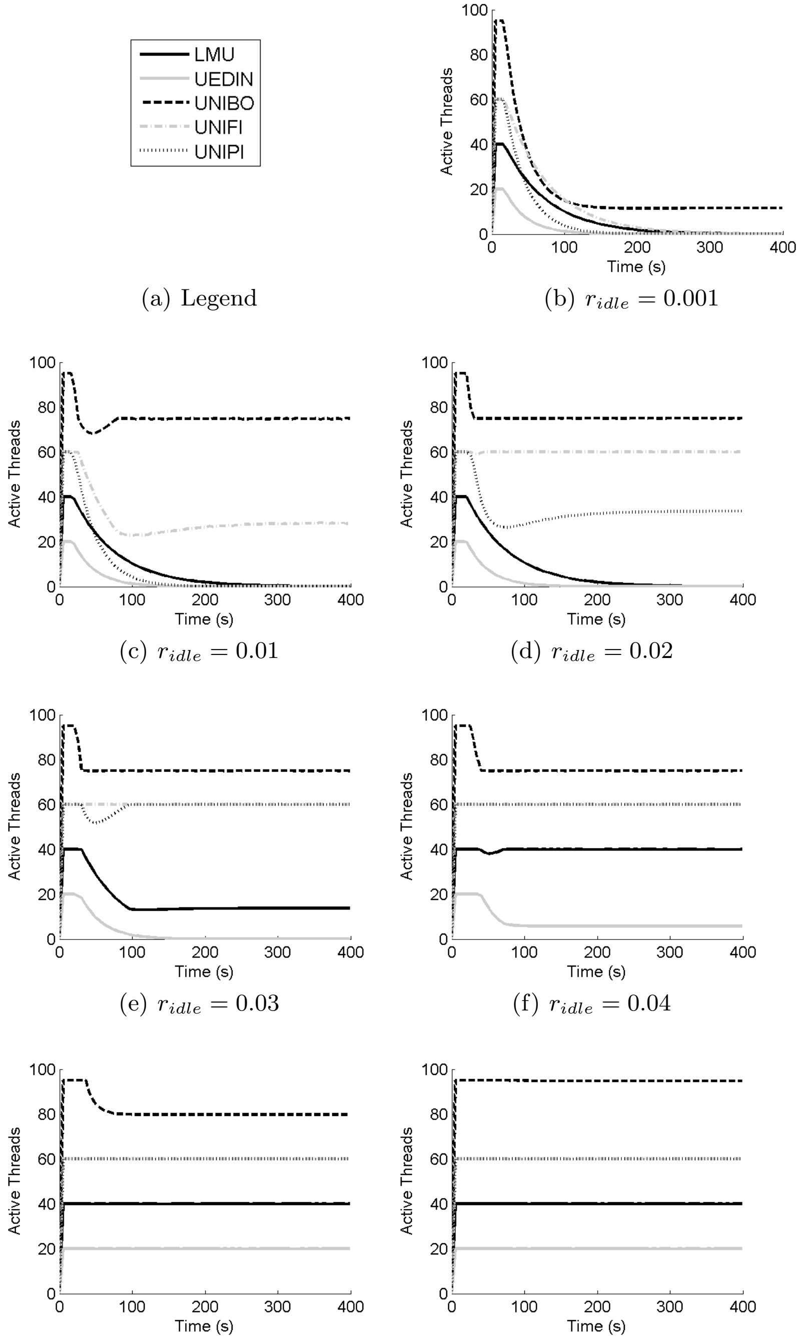
This model can be analysed through the underlying CTMC, stochastic simulation or ODEs. As far as Markovian analysis is concerned, the model allows us to fully take advantage of state space reduction by aggregation [13]. For instance, the state $[\mathit{Mirror}_{\mathrm{UNIBO}} = 74, \mathit{MirrorUploading}_{\mathrm{UNIBO}} = 1]$ aggregates 75 states. However, basic combinatorics suggests that the state space size of the underlying Markov chain, even with aggregation, is at least the product of the maximum number of incoming connections in each site. With this model's parameters, that would mean a Markov chain with over 273 million states, which is at the limit of the state-of-the-art in Markov chain solution technology. On the other hand, the model can be represented by a system of 17 coupled ODEs. This is the mathematical representation that we use to evaluate the performance of this system [5].

In this section we carry out time series analysis which allows us to see how the number of each type of component in the model varies as a function of time. This can provide the modeller with insights into the utilisation of the mirrors in both the transient and the steady state (as time increases and the transient behaviour tends to the equilibrium behaviour). In particular, we studied the impact that the client's behaviour has on such a performance index. The model parameters are as follows. The initial population of clients is 400. The deployment vector is the maximum number of available threads at each site as inferred from the definitions of the functional rates. Connection rate to all the mirror sites is 20.0. Available bandwidth per thread is 1/60 at LMU and UNIFI and 1/30 at UNIBO, UEDIN, and UPISA. We conducted sensitivity analysis of the *idle* activity by solving the system for the following values of r_{idle}: 0.001, 0.01, 0.02, 0.03, 0.04, 0.05 and 0.06. The graphs in Fig. 3 show time-series plots of the number of threads at each site in the 0–400 s time interval for such values of r_{idle}. The results were obtained by running the model through the adaptive step-size 5$^{\mathrm{th}}$-order Dormand Prince solver with default settings in our software tool, the "PEPAto" library [14] (100 data points, 0.001 step size, 1E-4 absolute error, 1E-4 relative error).

We compared the results from the numerical integration of the differential equations against stochastic simulation. Figure 4 shows good agreement between the deterministic trajectory (black line) and four independent runs of Gillespie's stochastic simulation algorithm (grey lines). The plot shows the evolution of the number of active threads at UNIFI for $r_{idle} = 0.01$. Similar fitting has been observed in the other cases under study.

6.1 Commentary on the results

From these analysis results we are able to see how the load on each server varies as a function of time and see how the speed with which all servers reach saturation varies as a function of variation in idle time. In Figures 3(b)–3(h) we see how the load on the servers is balanced out in response to increasing client demand. In our model increasing client demand is achieved by decreasing client idle time (going from $r_{idle} = 0.001$ to $r_{idle} = 0.06$). At the system initiation all clients stand ready to connect and so the load on the Bologna server (UNIBO) rises rapidly. Thus we are considering here a difficult case for the system, but one which is likely to occur in practice. In systems with large numbers of clients one often observes the well-known "flashcrowd effect" where large numbers of



(a) Legend

(b) $r_{idle} = 0.001$

(c) $r_{idle} = 0.01$

(d) $r_{idle} = 0.02$

(e) $r_{idle} = 0.03$

(f) $r_{idle} = 0.04$

(g) $r_{idle} = 0.05$

(h) $r_{idle} = 0.06$

Fig. 3. Utilisation of the mirror sites.

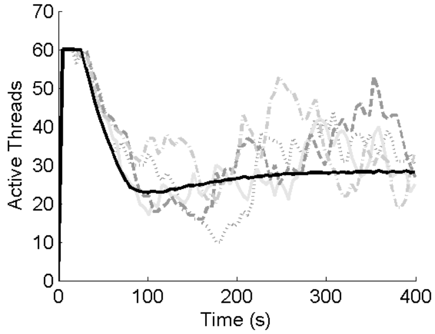


Fig. 4. Comparison between ODE analysis and stochastic simulation of the evolution of the number of active threads at UNIFI for $r_{idle} = 0.01$.

clients attempt to connect at the system initiation. This phenomenon is widely observed in peer-to-peer systems [9].

When the system is lightly loaded (Figure 3(b)) then after the initial flurry of work we find that from time 200 onwards the Bologna server is processing all requests itself and passing nothing on to the other servers. As the load increases (Figure 3(c) and (d)) we observe that the Bologna server is passing work on to the other servers in Italy (UNIFI and UNIPI). Small increases in load beyond this point cause work to be passed to the further-away Munich server (LMU) until it saturates (Figure 3(f)), and the Edinburgh server similarly (Figure 3(g)). Finally, the Bologna server must bear the remaining load itself (Figure 3(h)). These results show the load-balancing function at work in practice.

7 Conclusions

By federating the resources of the SOCK and PEPA process calculi we have been able to consider our case study of a replicated Web Service from both the functional and the non-functional (performance) perspectives. In a previous study we used analysis of a process calculus model using differential equations [5] to show that an architecture based on a centralised single server would not scale in the way desired [11]. In the present paper we use these methods to show that a replicated design does scale adequately. We have been able to use the continuous-space methods of [5] to analyse a model of a size which would defeat discrete-state analysis. The method is illustrated on the example of an e-learning system here but is generally applicable to analyse the scalability of replicated Web Services.

Acknowledgements: The authors are supported by the EC-funded FET-IST GC2 project number 016004 SENSORIA (Software Engineering for Service-Oriented Overlay Computers). The Jolie interpreter and the example considered here are available for download from http://jolie.sourceforge.net. The authors thank the anonymous reviewers for their insightful remarks which helped us to improve the paper for this version. Thanks to Adam Duguid for many helpful suggestions on model analysis.

References

1. Guidi, C., Lucchi, R., Gorrieri, R., Busi, N., Zavattaro, G.: Sock: A calculus for service oriented computing. In: Service-Oriented Computing - ICSOC 2006, 4th International Conference, Chicago, IL, USA, December 4-7, 2006, Proceedings. Volume 4294 of LNCS. (2006) 327–338
2. Busi, N., Gorrieri, R., Guidi, C., Lucchi, R., Zavattaro, G.: Choreography and orchestration conformance for system design. In: Conference on Coordination Models and Languages (COORDINATION'06). LNCS, Springer (2006)
3. Montesi, F., Guidi, C., Lucchi, R., Zavattaro, G.: JOLIE: a Java Orchestration Language Interpreter Engine. In: Proceedings of CoOrd 2006, ENTCS (2006)
4. Hillston, J.: A Compositional Approach to Performance Modelling. Cambridge University Press (1996)
5. Hillston, J.: Fluid flow approximation of PEPA models. In: Proceedings of the Second International Conference on the Quantitative Evaluation of Systems, Torino, Italy, IEEE Computer Society Press (2005) 33–43
6. Holton, D.: A PEPA specification of an industrial production cell. The Computer Journal **38**(7) (1995) 542–551
7. Forneau, J., Kloul, L., Valois, F.: Performance modelling of hierarchical cellular networks using PEPA. Performance Evaluation **50**(2–3) (2002) 83–99
8. Razafindralambo, T., Valois, F.: Performance evaluation of backoff algorithms in 802.11 ad-hoc networks. In: PE-WASUN '06: Proceedings of the 3rd ACM international workshop on Performance evaluation of wireless ad hoc, sensor and ubiquitous networks, New York, NY, USA, ACM Press (2006) 82–89
9. Duguid, A.: Coping with the parallelism of BitTorrent: Conversion of PEPA to ODEs in dealing with state space explosion. In: FORMATS 2006, Springer LNCS 4202 (2006) 156–170
10. Bradley, J., Gilmore, S., Hillston, J.: Analysing distributed Internet worm attacks using continuous state-space approximation of process algebra models. J. Comput. System Sci. (2007) doi:10.1016/j.jcss.2007.07.005. To appear.
11. Gilmore, S., Tribastone, M.: Evaluating the scalability of a web service-based distributed e-learning and course management system. In Bravetti, M., Núñez, M.T., Zavattaro, G., eds.: Third International Workshop on Web Services and Formal Methods (WS-FM'06). Volume 4184 of Lecture Notes in Computer Science., Vienna, Austria, Springer (2006) 156–170
12. Hillston, J., Kloul, L.: An efficient Kronecker representation for PEPA models. In de Alfaro, L., Gilmore, S., eds.: Proceedings of the first joint PAPM-PROBMIV Workshop. Volume 2165 of Lecture Notes in Computer Science., Aachen, Germany, Springer-Verlag (2001) 120–135
13. Gilmore, S., Hillston, J., Ribaudo, M.: An efficient algorithm for aggregating PEPA models. IEEE Transactions on Software Engineering **27**(5) (2001) 449–464
14. Tribastone, M.: The PEPA Plug-in Project. In: Proceedings of the 4th International Conference on the Quantitative Evaluation of Systems (QEST'07), IEEE Computer Society Press (2007) 53–54