A Lightweight Method for Analysing Performance Dependencies Between Services

Arjan Lamers[1,2] and Marko van Eekelen[2,3](✉)

[1] First8 BV, Nijmegen, The Netherlands
a.lamers@first8.nl
[2] Open University of the Netherlands, Heerlen, The Netherlands
[3] Radboud University Nijmegen, Nijmegen, The Netherlands
marko@cs.ru.nl

Abstract. For many applications, performance is paramount. For example, to improve conversion rates in e-commerce applications or to comply with service level agreements. Current trends in enterprise level architecture focus on designing and orchestrating *services*. These services are typically designed to be functionally isolated from each other up to a certain degree. During the design phase as well as when the application is deployed, choices have to be made how services interact and where they need to be deployed. These choices have a profound impact on the responsiveness of an application as well as on which performance can be made. In this paper we propose a methodology to describe and analyse performance dependencies between services. The resulting model can then be used to assist in designing a service oriented architecture and improving existing solutions by pointing out performance dependencies of services.

Keywords: Services · Deployment · Architecture · Design

1 Introduction

Current trends in enterprise level architecture are focused on delivering true components. Service Oriented Architecture (SOA) and Microservices are trends that aim at delivering components (*services*) [7,8,14,16] that can be used as ready-made parts. Building software products should then become a matter of orchestrating these services. A service in SOA is defined by OASIS [14] as *a mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description*. Typically services are grouped together in a *domain* and each domain is isolated to some degree from other domains. This degree of isolation can mean that different domains are managed by different companies or departments, that they are hosted in different data centres, on different machines or that they don't share e.g. the same database schema. This degree of isolation has profound impact on the resulting software

© Springer International Publishing Switzerland 2016
A. Celesti and P. Leitner (Eds.): ESOCC 2015 Workshops, CCIS 567, pp. 93–110, 2016.
DOI: 10.1007/978-3-319-33313-7_7

product: it impacts how software changes can be managed but also has an effect on performance issues [11] such as latency and scalability.

The main goal of the methodology described in this paper is to analyse performance dependencies of services in an architecture. In general, not all parts of an architecture have the same performance constraints. Some areas can be more focused on latency, others on throughput. Some services may be governed by a service level agreement (SLA) while others are less business critical. If parts of a system can be isolated and have their own constraints, the resulting product can be simpler and cheaper. Moreover, to be able to guarantee that a specific part of a system has a certain level of performance, it cannot be influenced by parts of the system that are not under full control (e.g. public API's). This method does not attempt to quantify performance aspects; doing so would require a detailed knowledge on the actual implementation. These are either not yet known (in the design phase) or prone to change due to functional or hardware changes. Furthermore, not all services may be owned by the owner of the architecture or are exposed to third parties. For example, traffic on public API's might be possible to be estimated, but in the event of marketing campaigns or DDOS attacks, these averages are not representative anymore. In these events, even a low order relationship between the API and other services might still be enough to degrade the system performance as a whole. It emphasises latency (responsiveness) although it reveals information about throughput bottlenecks as well.

The method has been used by the author at different stages of various projects. During design it helped to determine how to design interaction between services as well as to define domain boundaries. It was also helpful with investigating performance problems in an existing architecture. Proposed changes were again validated using the model. Fellow architects in those projects were able to quickly explain performance issues more concisely using the methodology without having to invest in expensive tests or complex modelling. Business stakeholders were able to understand performance consequences of decisions and understand the reasoning behind the proposed changes. The methodology gives a more concise and formal output than the 'gut feeling' that a proposed change might improve the performance of an architecture.

The model assumes a given set of services. Higher level abstractions such as processes, or lower abstractions such as components are all flattened to basic services. By modelling the way the services interact it is possible to predict potential performance issues and solve them. It also helps in determine which services can be grouped together from performance point of view and as such can help in (re)defining domains. The interaction between services is described by making a distinction between a flow of information and a flow of initiative: is the information pushed or pulled? Rather than focusing on describing an algorithm or optimising a protocol between services, the method focuses on questioning if two services should be connected at all and, if so, which service should take initiative. In any sufficiently complex architecture, information can take a significant amount of time to travel through the system. Optimising that information

latency while at the same time managing performance constraints is not trivial. The method first focuses on trying to solve this issue on the architecture level. The proposed abstraction is simple enough to allow discussion between software architects and domain experts, negotiating on performance aspects, while still expressive enough to meaningfully guide an architecture. Local optimisations can follow afterwards.

The methodology consists of three steps. First an *architecture* is defined in Sect. 2. This describes the services and their interactions. Also, isolation constraints can be formulated. Next, these services have to be run on machines, potentially having more than one instance of a service. This is described in the *deployment allocation* (Sect. 3) of the model. In Sect. 4 managing state is discussed. Based on the resulting optionality of connections, this deployment allocation can be configured by choosing which connections between machines are optimal. This results in a *deployment configuration* as described in Sect. 5. In each step, the isolation constraints of a service can be verified.

2 Architecture Layer

2.1 Service Interaction

In SOA, services are consumed by work flows or processes. Services can also be composed out of other services, making the model fractal. In this methodology, everything is flattened to a service. If a service consists of components that can be deployed by themselves (e.g. a service using a database), those components are considered services as well.

A service is considered a vertex in a graph. The edges represent *calls* from one service to another. There are two properties to be considered when describing interaction between two services. The first property is the flow of information, the second defines which service takes initiative. If service a has information that is required by service b, that information can be *pushed* from a to b (Fig. 1).

Fig. 1. a pushes to b

Fig. 2. b pulls from a

The initiative can also originate from service b. In this case, a is *pulled* by b (Fig. 2). Information still flows in the same direction, but the initiative is placed with the receiver instead of the sender.

A push from a to b is considered a fire-and-forget operation. It is assumed that even if b is busy, a can continue its work without significant delay. If a confirmation of a push which can have a significant delay (e.g. the confirmation contains a business result) is required information flows back from b to a. Therefore, an additional edge is required: a pulls from b.

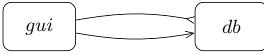

Fig. 3. Gui and db

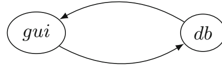

Fig. 4. Information graph

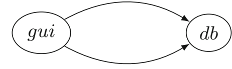

Fig. 5. Initiative graph

As an example throughout the paper, consider an online bookstore. In its basic form, it consists of a web application consisting of a graphical user interface (gui) and a database (db), as shown in Fig. 3.

In this scenario, the user interface allows users to enter new information or change information in the database. Thus, it *pushes* information as entered by the user to the database. The user interface also can query the database so it also *pulls* information from the database. Information flows in both directions (Fig. 4). All initiative, however, always originates from the user interface. In other words, this is a classical client-server setup; without a client, the server (db in this case) has nothing to do (Fig. 5).

Formally, an *architecture* A consists of a vertex set S for the services and an edge set C^A representing the *calls* between services. A typed edge (p, s, t) with $p \in \{push, pull\}$ going from source s to target t is defined as having s as the source of information. Thus, in the example above, $S = \{gui, db\}, C^A = \{(push, gui, db), (pull, db, gui)\}$.

This model can be translated to two different graphs, an information flow graph and an initiative graph:

The information graph I_{inf} for a given architecture A is defined as:

$I_{inf}(A) = (S, E)$, where S is the same service set of A. The edge set E is defined as: $E_{inf}(A) = \{(s, t)|(p, s, t) \in C^A\}$.

The initiative graph I_{int} for a given architecture A is defined as:

$I_{int}(A) = (S, E)$, where S is the same service set of A. The edge set E is defined as: $E_{int}(A) = \{(s, t)|(p, s, t) \in C^A \wedge p = push\} \cup \{(t, s)|(p, s, t) \in C^A \wedge p = pull\}$, reversing the *pull* edges.

2.2 Stress and Responsiveness

In this model, a couple of properties can be defined.

The first property is *stress*. The *stress graph* (Fig. 6) of a service s is defined as the subgraph of vertices that can reach the service s in the initiative graph, including s itself. This means that the amount of work to be done on that service s is influenced by all the services in the stress graph. For the example above, the *stress set* STR of vertices in the stress graph are: $STR(gui) = gui, STR(db) = \{gui, db\}$. This can be interpreted as follows: an increased load on gui will lead to an increased load on db, but an increase in load on db does not lead to more load on gui.

More formally, for a service S in an architecture A, the stress set STR is defined as:

$STR(s) = \{s\} \cup STR_{push}(s) \cup STR_{pull}(s)$, with:

$STR_{push}(s) = \bigcup_{(p', s', t') \in C^A} \{STR(s') | s' = s \wedge p' = push\}$

$STR_{pull}(s) = \bigcup_{(p',s',t') \in C^A} \{STR(t') | t' = s \wedge p' = pull\}$

To determine if a service can quickly respond to a request, the above properties are not enough. The stress indicates what impacts resource usage but the service might also require information from another service. If the database has a high load, it will still impact the user interface: retrieving information will be slower. To represent this a second property is introduced, *responsiveness* (Fig. 7), combining the stress on the service s with the stress of the services from which it *pulls*. For the example above, the responsiveness RES is $RES(gui) = RES(db) = \{gui, db\}$.

The *responsiveness set* (Fig. 7) is thus more formally defined as follows: $RES(s) = STR(s) \cup \bigcup_{(p',s',t') \in C^A} \{RES(s') | p' = pull\}$

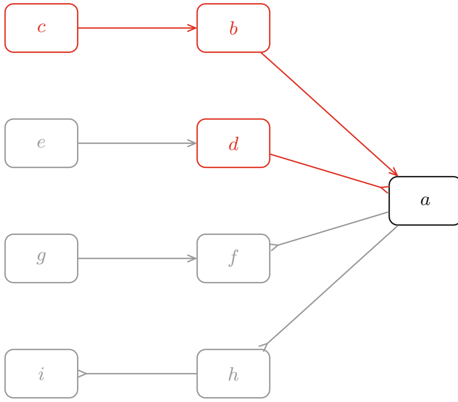

Fig. 6. $STR(a) = \{a, b, c, d\}$

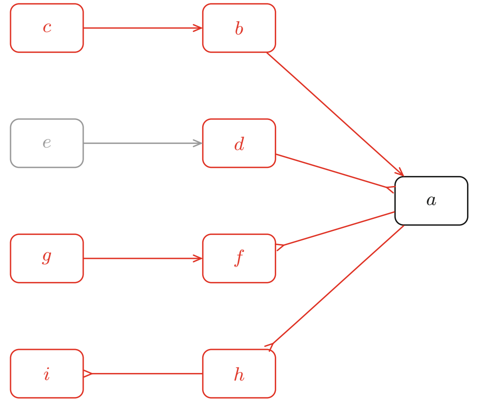

Fig. 7. $e \notin RES(a)$

2.3 Analysing an Architecture

In any architecture, different services have different non-functional requirements. Typically, a user interface has to respond quickly to an end-user's actions. The model cannot give a quantitative measurement. However, it can reveal which services impact the user interface. This allows us to define constraints that enforce a disconnect in performance between two services. For example, consider a system with, amongst other services, a user interface and a public API. A typical constraint might be that the user interface should always be responsive, no matter the (uncontrolled) load on the public API. This constraint can be proven in the model by showing that the public API is not in the responsiveness set of the user interface.

We can thus define the following two constraints:

Definition 1. *A service a is* weakly isolated *from b if $b \notin STR(a)$.*

Definition 2. *A service a is* strongly isolated *from b if $b \notin RES(a)$.*

3 Deployment Allocation

The next step is to describe the machines on which the services will be deployed. Larger systems might require multiple (virtual) *machines*. If multiple machines are available, the option arises to deploy services isolated on machines or to combine a subset of them on a single machine. A service can even be deployed multiple times to be able to handle more traffic.

If two services are deployed on the same machine, they will share resources and thus their *stress* will be shared. A perfectly scalable architecture might thus be deployed in such a way that it looses its responsiveness properties. On the other hand, deploying two communicating services on separate machines will introduce additional network latency. Furthermore, since it is typically assumed that a network might fail, the two services will have to deal with CAP problems. To analyse this, a deployment layer will be added to the model.

A *deployment allocation* for an architecture A is a set of *machines* with each machine running a subset of A's services. The *calls* made between services in A are expanded to *connections* in the deployment: for each *call* from s to t in the architecture, a similar *connection* is made between every machine that runs s and every machine that runs t (Fig. 8).

Back to the bookstore example application. Initially, it might be deployed on a single machine, running both the *gui* and the *db*. If the bookstore is successful, the traffic to the website will increase. At some point, the single machine does not have enough resources to manage the traffic. Typically, the easiest way to scale up is to *vertically scale* by buying bigger hardware, or dividing the services over multiple machines as in Fig. 9. A next step, assuming that the bottleneck is the *gui* as it often is, could be to *horizontally scale* by deploying more than one instance of a service on multiple machines, as seen in Fig. 10.

In this example, machine *1* and *2* contain a deployment of the service *gui*. Machine *3* contains a deployment of the service *db*.

A deployed service σ for service s on machine m is defined as $\sigma = (s, m) \in \Sigma$. A deployment allocation D is graph with vertex set Σ of deployed services and an edge set of connections C^D.

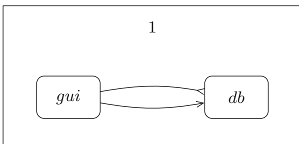

Fig. 8. 1 = {gui, db}

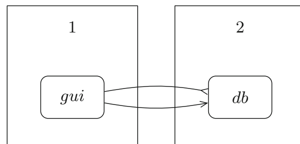

Fig. 9. 1 = {gui}, 2 = {db}

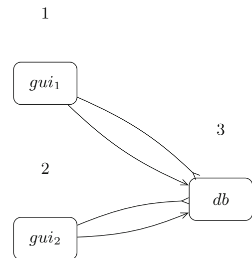

Fig. 10. 1 = 2 = {gui}, 3 = {db}

For convenience, the set of machines within D is defined as $M = \{m|(s,m) \in \Sigma\}$. The *deployment set* of a service s is defined as the set of machines that deploy s: $DEP(s) = \{m|(m,s) \in \Sigma\}$.

A machine holds a subset of services and all services should be deployed: $\forall(s,m) \in \Sigma : s \in S$ and $\forall s \in S : \exists(s,m) \in \Sigma$.

The edge set is derived from the services and holds a reference to the original call:

$C^D = \bigcup_{(p_a,s_a,t_a)\in C^A}\{((p_a,s_a,t_a),s_d,t_d)|s_d = (s_a,m) \wedge m \in DEP(s_a), t_d = (t_a,n) \wedge n \in DEP(t_a)\}$.

3.1 Analysing a Deployment Allocation

Similar to the properties STR and RES as defined in the context of an architecture, we can define analogue properties for machines and services in the context of a deployment. Services deployed on the same machine share resources such as memory or cpu. Therefore they share stress.

The isolated stress of a deployed service (s,m), i.e. the stress without considering other services on the same machine, is defined in a similar way to the stress of a service:

$str(\sigma) = \{\sigma\} \cup strpush(\sigma) \cup strpull(\sigma)$
$strpush(\sigma) = \bigcup_{((p',s',t'),\sigma',\varsigma')\in C^D}\{str(\varsigma')|\sigma' = \sigma \wedge p' = push\}$
$strpull(\sigma) = \bigcup_{((p',s',t'),\sigma',\varsigma')\in C^D}\{str(\varsigma')|\varsigma' = \sigma \wedge p' = pull\}$

The actual stress of a service s on machine m is thus simply the stress of machine m, $STR(s,m) = STR(m) = \bigcup_{(s',m)\in D} str((s',m))$.

We can now also define the responsiveness of a service *on a machine*. While stress is automatically shared between services (since they share resources), the responsiveness of two services on a different machine might still be different since they can pull from different sources. The responsiveness of a service s on the machine m is thus defined as the stress of the machine m united with the responsiveness of all deployed services that are pulled from service s. More formally,

$RES(s,m) = STR(m) \cup \bigcup_{((p',s',t'),\sigma',\varsigma')\in C^D}\{RES(\sigma')|p' = pull\}$

We can thus redefine the isolation constraints on deployment allocation level:

Definition 3. *A service a is* weakly isolated *from a service b if* $\forall_{m\in DEP(a)}\forall_{(s',m')\in STR(m)} s' \neq b$.

Definition 4. *A service a is* strongly isolated *from a service b if* $\forall_{m\in DEP(a)}\forall_{(s',m')\in RES(m)} s' \neq b$.

As an example, consider Fig. 10 again. The *gui's* have been horizontally scaled, but how effective is that? The stress of the machines in this example is:

$STR(1) = \{(gui_1,1)\}, STR(2) = \{(gui_2,2)\}, STR(3) = \{(gui_1,1),(gui_2,2), (db,3)\}$

The responsiveness of the deployed services are:

$RES(gui_1,1) = RES(gui_2,2) = RES(db,3) = \{(gui_1,1),(gui_2,2),(db,3)\}$. The stress property indicates that all machines provide stress on the db, making

it a likely bottleneck. Furthermore, while the *gui* deployments don't share stress, they still influence each other in responsiveness: if one *gui* puts a high load on the *db*, it will impact the other *gui*'s responsiveness.

Thus, both *gui* services are only weakly isolated from each other, not strongly isolated.

There are also some new properties to be discussed. In a distributed deployment (a deployment with $|M| > 1$), communication between two machines is done via network calls. These are orders of magnitude slower than local calls. Therefore, to reduce latency in a system, it is necessary to reduce the number of network hops. Secondly, since network connections are more prone to break, it is more important to define a consistency model which allows for faulty communication channels. To avoid network hops, one could collocate two services on the same machine. This, however, will result in them sharing stress.

A network hop or *non-local connection* is a connection that has its source and target services on different machines.

Thus for a connection c, with $c = (c', m', n')$:

$$local(c) = \begin{cases} 0 & \text{when } m' = n' \\ 1 & \text{when } m' \neq n' \end{cases}$$

There are two important properties that are impacted by the network hops. Firstly, the responsiveness is not only impacted by stress on the machines, network latency is an important factor as well: *responsiveness network depth.* The responsiveness network depth $RNET(s, m)$ for a service s on a machine m is defined as the maximum number of network hops to any other service which can be reached by s via *pull* requests. Note that if there are cycles in the graph, the network depth is defined to be infinite. More formally, assume a *pull-graph* P for (m, s) is a weighed graph derived from a deployment D with the same vertex set M. The edge set for P is defined as all the pull edges for s as well as all pull edges for the source vertices of those edges. An edge c in the pull-graph derived from edge c' in the deployment has weight $local(c')$. $RNET(s, m)$ is now the maximum of the sum of weights of each branch from s. If the graph is not a tree, $RNET(s, m) = \infty$.

Secondly, to accurately define a consistency model allowing for failing network connections, one needs to take into account the full source of information: *consistency network depth.* The first property to discuss is *consistency.* As with any system, there is a delay whenever information is passed from one point to another. As such, for a service to have a world view on its state consistent with the whole chain, any and all change in information it requires has to have reached the service. The subgraph of all vertices that can reach a service s in an architecture, including s itself, in the information graph is defined as the *consistency graph* for s. The *consistency set* for s is the set of vertices within the consistency graph. For the group of services in this set, consistency model limitations will hold (e.g. CAP limits). Either these services are deployed on a non-partitionable system, or availability/consistency limitations will arise. More formally, for a service s in an architecture A, the consistency set CON is defined as the set of vertices including s that can reach s in the information graph $I_{inf}(A)$:

$CON(s) = \{s\} \cup \bigcup_{(p',s',t') \in C^A} \{CON(s') | s' = s\}$

Similar to $RNET$, the *consistency network depth* $CNET(s, m)$ is defined as the maximum number of network hops to any service that provides information for s. The same definition applies, only using the information graph $I_{inf}(A)$ instead of the pull graph.

These properties can be used to analyse and reduce the number of network calls for a specific service. $CNET$ gives an indication from *how far* information has to come, thus increasing consistency model complexity, whereas $RNET$ indicates how much the network impacts the responsiveness.

4 State

When distributing a service, there is always the matter of synchronizing state. Changes in one instance of a service might impact another instance of a service. This impacts how an application can be deployed and which calls and connections are required. To reflect this, the model supports three kinds of *statefulness* for a service, *stateless*, *stateful* and *partitionable.*

These are defined as follows:

A service is *stateless* if, when there are multiple instances deployed of that service, they do not require any exchange of information between those instances to be able to fulfil all requests. In other words, each instance can be deployed fully isolated while still be able to serve all requests.

A service is *partitionable* if, when there are multiple instance deployed of that service, a specific instance can handle the request in isolation. The instance that is able to handle a specific call must be determined based on the content of that call. Each instance holds its own subset of the state and can manage that independently. A call is called *routable* if the correct instance can be determined based on the parameters of the call.

A service is *stateful* if, when there are multiple instance deployed of that service, they do require to synchronise state in order to be consistent with each other.

Recall that an architecture A has a vertex set of services S. A service $s \in S$ with name n and statefulness p is defined as a tuple (n, z) with $z \in \{stateful, stateless, partitionable\}$. For a partitionable service, it is further relevant on which dimensions it can be partitioned. To simplify, it is assumed there is only a single dimension on which a service can be partitioned if it is partitionable. If $s = (n, z), z = partitionable$ then its partitioning dimension should also be defined as $PART(z)$.

In the bookstore example, the architecture could be further refined to include an explicit business layer service. This might be deemed necessary due to an increase in features or due to a need for different front ends. The bookstore's architecture will then look like the following classical 3-tier architecture:

The graphical user interface can be scaled to have multiple instances. When a user logs in, he or she has a session at a specific instance and as such all requests related to that session can be managed by that single instance in isolation. Thus, the gui is partitioned by sessions.

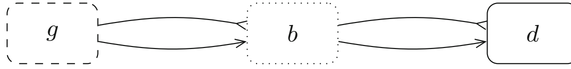

Fig. 11. Partitioned gui (g), stateless business layer (b) and stateful database (d)

The business layer handles requests from the gui, interprets them, applies business rules and uses the database to store information. It does not keep any state between calls so if there are multiple instances of the business layer service, they can act isolated. The business layer is thus stateless.

The database stores the information as requested by the business layer. If there is more than one instance, these instances need to be synchronized in order to stay consistent. Thus, the database is stateful.

4.1 Deploying with State

When an architecture is deployed, the *statefulness* of a service determines how it affects the different connections resulting from calls between services. A call can be given as either *routable* or a *broadcast* in this model. Routable means that a single instance of a service is sufficient to handle the request and that it is known which instance this is. A broadcast means that all instances of a service need to handle the request. A routable call to a stateless or stateful service means that the request can be handled locally, a broadcast to these services is probably a cache invalidation or some other global effect. A routable call to a partitioned service means that by the nature of the request or its payload it can be determined which partition holds the subset of data required to process the request. A broadcast is necessary if it is unknown which partition holds the data, or all partitions are required to process the request. Graphically, this is indicated by the color of the service s (based on $PART(s)$) and the color of the connection (again based on $PART(s)$) where s is the source in the information graph.

Recall that an architecture A has an edge set C^A with a call $c = (p, s, t) \in C^A$. To represent the *routability* property, the tuple is redefined as $c = (p, s, t, r) \in C^A$ where $r \in \{routable, broadcast\}$.

When there is more than one instance of a stateful service, these instances need to synchronize. For that to happen, information has to be exchanged and that means that broadcast calls between all instances exist. By convention, these calls are designated as broadcasting push connections. For the properties as defined until now, it does not matter if it is push or pull since the call is from a service to itself. To indicate if a call is a broadcast or if it is routable, respectively double and single arrow heads are used in an architecture graph as in Fig. 12.

In extending a deployment graph from an architecture graph, connections are derived from calls. The connections will derive a new property *optionality* which can be *deployment optional*, *runtime optional* or *compulsary*, based on the type of service and if the call is routable or not.

A connection is by definition *compulsary* if the call is a broadcast, since all instances of a service have to be reached.

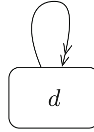

Fig. 12. Stateful database (d) needs to synchronize using a broadcast

A connection is *deployment optional* when, for all the connections in a deployment for a specific call, only one is necessary for the system to function correctly. All others can be left out of the deployment. If the target of a call in the initiative graph is either stateless or stateful, any of the deployed instances can be the target for the connection. Each instance can handle the request. This type of connection is considered *deployment optional*. A push to or a pull from any stateless or stateful service is considered deployment optional by default.

A connection is *runtime optional* when, for all the connections in a deployment for a specific call, only one connection is used in a specific instance. Which one it is, is determined at runtime. Other connections may be used for different calls. If the target of a call in the initiative graph is partitioned and the call is routable, only one connection is used runtime to the specific instance of the partitioned service. This type of connection is considered *runtime optional*.

A connection $c \in C^D$ is now defined as $c = (c', \sigma, \varsigma, o)$, with as before $c' \in C^A$ and having a deployed service $\sigma \in \Sigma$ as a source of information and $\varsigma \in \Sigma$ as a target. The new property $o \in \{runtime - optional, deployment - optional, compulsary\}$ is added.

As an example, consider deployments for the architecture as defined in the bookstore's 3-tier architecture (Fig. 11). If two instances for each service are created, the deployment as seen in Fig. 13 is the result. Here the dotted lines are deployment optional, the solid lines are compulsary. The two database instances are synchronized in what is generally called a master-master replication. Other database replication scheme's would require a change in architecture first. For

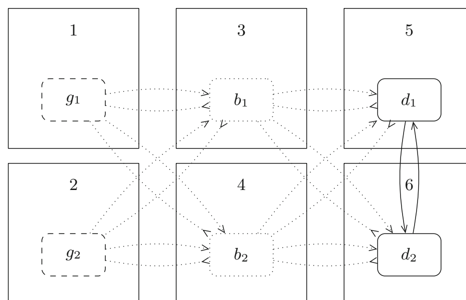

Fig. 13. Deployed partitioned gui (g), stateless business layer (b) and stateful database (d)

example, read-only slave configurations require that the client (in this case the business layer) knows which database to use for writes and which for reads. Thus, without changing the architecture, this is the resulting deployment.

If the statefulness of the business layer is changed, the connections will change as well. For example, assuming the business layer is partitioned as well, the graph will look like Fig. 14, where the dashed lines represent the runtime optional connections. Imagine for example that each business layer instance services different payment options (e.g., mastercard transactions go to b_1 and visa to b_2). In this example, the g and b services use different partitioning dimensions ($PART(g) \neq PART(b)$): the gui is partitioned by user sessions whereas the business layer by payment options. The connections from g to b thus have to be routable on $PART(b)$. As a last example, if the statefulness of the business layer is stateful, the graph will look like Fig. 15. This is quickly the case if the business layer manages its own state instead of delegating to the database.

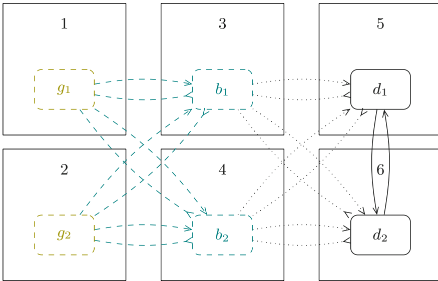

Fig. 14. Partitioned business layer

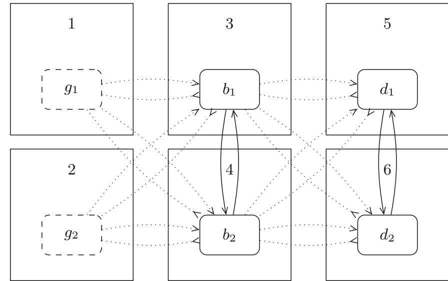

Fig. 15. Stateful business layer

5 Deployment Configuration

The deployment allocation assignes services to machines. Given a deployment allocation, the optionality of the connections between machines is known and some of those connections are redundant. Based on this, non optimal connections can be pruned and configuration choices can be made. Some of these choices are obvious improvements, while others have both advantages and disadvantages. Choosing which connections to actually configure results in a *deployment configuration.*

5.1 Deployment Optional Pruning

In case of deployment optional connections, if one of the connections is *local* than that one is generally preferred; there is no obvious reason to use a non-local connection since all are equal. By picking the local connection, that connection is no longer deployment optional, there is nothing else to choose from. For example, considering the bookstore 3-tier architecture (Fig. 11). Due to budget constraints

or other reasons, the gui and the business layer are to be deployed together on the same machine, resulting in a deployment allocation that will initially look like the graph in Fig. 16.

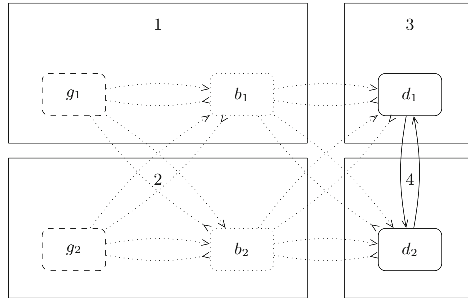

Fig. 16. Initial

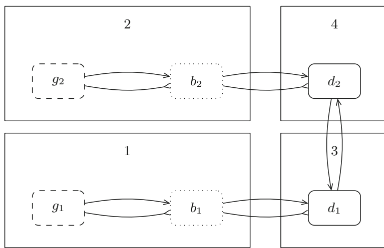

Fig. 17. Static load balancing

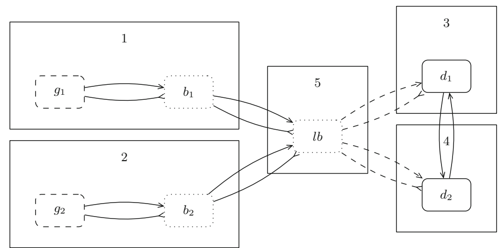

Fig. 18. Dynamic load balancing

The connections on the left side are all deployment optional and result from g and b having multiple instances. As such, all possible connections are derived from the architecture into the deployment allocation. However, since only one connection for each call is required, non-local connections can be removed, avoiding network calls when not required. For the deployment optional calls between the business services b_1 and b_2 to the databases instances d_1 and d_2, a couple of options are possible. One obvious choice is to assign each business service its own database. This would lead to Fig. 17 with each business service having compulsary connections to a dedicated database. Another is to dynamically load balance request between the databases. That would require an additional load-balancer service (lb) which routes the traffic to one of the database instances (Fig. 18). The connections from the business service to the loadbalancer become compulsary, while the connections from the loadbalancer to the database will be runtime optional; only one is required. While having a load balancer might lead to a more evenly distributed load over both database instances, the load balancer by itself is another bottleneck and network hop.

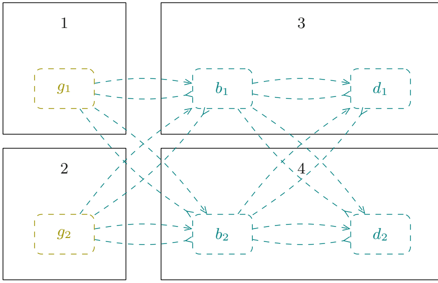

Fig. 19. Initial

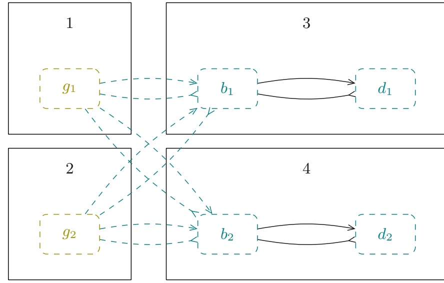

Fig. 20. Pruned as processing units

5.2 Pruning Runtime Optionals

For runtime optional connections resulting from partitioned services, some pruning options are possible as well. If both the source s and the target t of a connection are partitionable, share the same partitioning dimension and are always codeployed, the deployment configuration can exploit that by assigning the same partitions for instances of s and t. The assumption here is that a call does not change routing. If the use case requires a different routing, it should be marked as a broadcast. If this assumption holds, the machine which holds both s and t can be treated as a "processing unit" which deliver all functionality for a subset of partitions.

As an example, recall the partitioned business layer architecture in Fig. 14. Each business layer served a subset of payment methods (e.g. mastercard to b_1 and visa to b_2). It might be beneficial to partition the database in a similar way, storing only mastercard transactions in one instance, and visa transactions in the other. This way, both database instances can operate independently, resulting in the deployment allocation as found in Fig. 19. The consuming services, in this case a differently partitioned user interface g, should be able to route its calls to one of the processing units formed by machines 3 or 4. Pruned, this could be reduced to Fig. 20.

6 Architectural Patterns

To resolve performance issues, there are a number of technical patterns available that will isolate service performance to some degree. In this section common patterns like caches and queues are discussed, modeled and compared using the method. The most basic patterns, *push* and *pull* have been discussed in the first chapter as they are the building blocks of the model.

A *cache* pattern is used to keep state readily available if it has been calculated or received before. This way, the consumer is decoupled from the performance of the producer. A cache can behave in a *lazy* way, and only retrieve values when they are requested as modelled in Fig. 21. Here, the consumer c pulls the information from a cache store (*cdb*). If this store does not contain the value,

it retrieves it from the producer. While the *stress* of the consumer is decoupled from the producer, the model shows that the responsiveness is still dependent on the producer. In effect, the cache has no effect in the model since the producer and consumer are not fully isolated in the case of a cache miss.

Caches can also behave in an *eager* fetching way as modelled in Fig. 22. Here the *cdb* cache store is filled by an independent cache reader which pulls the original information from the producer. This can be a scheduled or an asynchronous task. In this scenario, the consumer's stress is isolated but the producers stress is depending on *cr*. The responsiveness of the consumer is now only dependent on *cr* and *cdb*. The *cr* service can thus be tuned to balance the stress on the producer versus the responsiveness of the consumer.

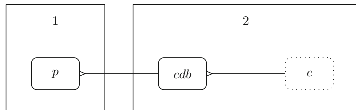

Fig. 21. Typical lazy cache

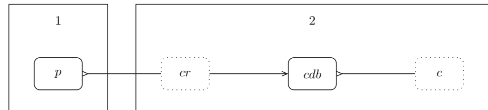

Fig. 22. Typical eager cache

A *queue* pattern is used to decouple a flow between services. One service pushes a message onto the queue, another service can pick it up at any time. See Fig. 23. The producer p can always deliver its messages and as such is unaffected by the performance of the consumer c. The model shows this as well: the stress and the responsiveness of p only depends on p itself. The consumer c also only receives stress from itself, but the responsiveness is impacted by both the queue (q) itself as well as the producer p. A queue reader or writer might be added (similar to the cache reader above, or even by adding a complete cache) to be able to improve responsiveness of the consumer.

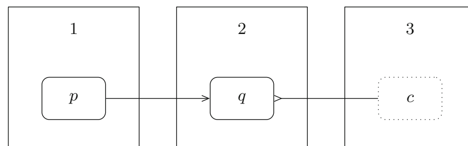

Fig. 23. A queue between p and c

In Table 1 presents a summary on how these architectural patterns behave according to this model. Note that the more performance isolation a pattern offers, the more elements are involved in maintaining consistency. For a queue, the consistency network depth also increases. As can be predicted, caches and push calls are excellent for improving responsiveness since they decrease the distance of accessing data (decreasing $RNET(s)$).

Table 1. Isolation levels of patterns

Pattern	p	c	$RNET(c, 2)$	$CON(c, 2)$	$CNET(c, 2)$
Push	Strong	-	0	{c,p}	1
Pull	-	Weak	1	{c,p}	1
Lazy cache	-	Weak	0	{c,p,cdb}	1
Eager cache	Strong	Strong	0	{c,p,cdb, cr}	1
Queue	Strong	Weak	1	{c,p,qdb}	2

7 Related Work

In this paper we have presented a novel notation. Other notations, such as UML sequence diagrams or Petri nets, also exist. Sequence diagrams can express parallelism and ordering of actions, expressing interaction between services quite detailed. Petri nets allow concurrency and synchronisation analysis in distributed systems and as such require details on how state is synchronised. These details are very useful *within* a specific service or domain but less useful *between* domains since these are, by definition, reasonably isolated. Instead, our notation leaves out algorithmic details and focuses on expressing the distinction between the source of information and initiative on a higher abstraction. This allow a focus on the question whether the architecture or the deployment needs to change or whether some latency requirements can be loosened, before trying to optimise it in the implementation.

Research has been done which focuses on predicting a quantified throughput of a (workflow in a) Service Oriented Architecture, e.g. [3,4,6,10,17]. In general, these models require load functions, detailed descriptions or actual implementations for each service. Determining load functions and finding reasonable values for parameters of these models can be quite demanding and might be possible only quite late in the development process. Additionally, calculating the performance of the architecture might not be instant but requires a (relatively) long simulation. Instead, our work focuses on finding performance isolation between services without quantifying it. The properties can be quickly derived, even manually up to a certain complexity, and future tooling could extensively compare alternatives. SLAng [15] provides a precise way of defining SLA's for services. It would be interesting to see if some properties could be guaranteed by the model.

Software defined networks [13] decouple the network control decisions from the actual hardware, making it easier to change deployment configurations, either manually or automatically.

8 Conclusion

The described method gives insight into how services influence each other with regards to performance. This can be used to validate and assist in decisions both

on architectural level as well as on deployment. Since the model does not require concrete details it can be used as a light weight method to drive discussion and validate performance requirements. Multiple implementations of a simple example, a bookstore website, were modeled and analysed, providing insight in difference in performance behaviour. Here the method provides a tangible result for performance related issues within an architecture. Possible solutions on both architectural (software) level as well as on deployment level can be compared and weighed.

Future Work

The methodology described can be applied to both small and larger architectures. For small architectures, this can be done by hand and the results are natural. For larger architecture tooling is required to derive results and these might be surprising. A tool is being build to automate calculation of the properties. This should aid in quickly discovering and analysing deployment scenario's and weighing the advantages and disadvantages such as balancing isolation versus network latency. It should also be able to point out possible areas where changes in the architecture could be beneficial and potentially detect (a subset of) performance anti-patterns [5]. Changing the initiative from one service to another, or edges that are suitable candidates for static or dynamic loadbalancing, could be auto detected and then alternatives could be compared. Other "Middlepipes" [9] related products such as circuit breakers as shown in e.g. [12] could be modelled as well, either as concrete specialisations or by deriving REO connectors [1]. The properties could be further formalised to derive optimisations for e.g. nested architectures and deployments. More research is to be carried out to see if we can help discover consistency models between services based on the initiative and information graphs, e.g. to help derive application invariants for [2].

To further validate the approach, the methodology should be applied at full scale projects in different stages of development or production.

References

1. Arbab, F.: Reo: a channel-based coordination model for component composition. Math. Struct. Comput. Sci. **14**, 329–366 (2004). http://journals.cambridge.org/article_S0960129504004153
2. Bailis, P., Fekete, A., Franklin, M.J., Ghodsi, A., Hellerstein, J.M., Stoica, I.: Coordination avoidance in database systems. Proc. VLDB Endow. **8**(3), 185–196 (2014). http://dx.org/10.14778/2735508.2735509
3. Bertoli, M., Casale, G., Serazzi, G.: JMT: performance engineering tools for system modeling. SIGMETRICS Perform. Eval. Rev. **36**(4), 10–15 (2009)
4. Brebner, P.C.: Real-world performance modelling of enterprise service oriented architectures: delivering business value with complexity and constraints. In: ACM SIGSOFT Software Engineering Notes, vol. 36, pp. 85–96. ACM (2011)
5. Cortellessa, V., Di Marco, A., Trubiani, C.: An approach for modeling and detecting software performance antipatterns based on first-order logics. Softw. Syst. Model. **13**(1), 391–432 (2014)

6. Ferrer, A.J., Hernández, F., Tordsson, J., Elmroth, E., Ali-Eldin, A., Zsigri, C., Sirvent, R., Guitart, J., Badia, R.M., Djemame, K., et al.: Optimis: a holistic approach to cloud service provisioning. Future Gener. Comput. Syst. **28**(1), 66–77 (2012)
7. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. Ph.D. thesis, University of California, Irvine (2000). aAI9980887
8. Fowler, M.: Microservices. http://martinfowler.com/articles/microservices.html
9. Jamjoom, H., Williams, D., Sharma, U.: Don't call them middleboxes, call them middlepipes. In: Proceedings of the Third Workshop on Hot Topics in Software Defined Networking, pp. 19–24. ACM (2014)
10. Kounev, S.: Performance modeling and evaluation of distributed component-based systems using queueing petri nets. IEEE Trans. Softw. Eng. **32**(7), 486–502 (2006)
11. Kratzke, N.: About microservices, containers and their underestimated impact on network performance. In: Proceedings of CLOUD COMPUTING 2015 (6th International Conference on Cloud Computing, GRIDS and Virtualization) (2015)
12. Netflix: Hystrix. https://github.com/Netflix/Hystrix
13. Nunes, B., Mendonca, M., Nguyen, X.N., Obraczka, K., Turletti, T., et al.: A survey of software-defined networking: past, present, and future of programmable networks. IEEE Commun. Surv. Tutorials **16**(3), 1617–1634 (2014)
14. OASIS: Oasis soa reference model tc. https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=soa-rm
15. Skene, J., Lamanna, D.D., Emmerich, W.: Precise service level agreements. In: Proceedings of the 26th International Conference on Software Engineering, pp. 179–188. IEEE Computer Society (2004)
16. The Open Group: Service oriented architecture: What is soa? http://www.opengroup.org/soa/source-book/soa/soa.htm#soa_definition
17. Zhu, L., Liu, Y., Bui, N.B., Gorton, I.: Revel8or: model driven capacity planning tool suite. In: 29th International Conference on Software Engineering, ICSE 2007, pp. 797–800. IEEE (2007)