Exploring Non-Functional Coupling Between Subsystems

Matt Pope

Brigham Young University

Follow this and additional works at: https://scholarsarchive.byu.edu/etd

Part of the Physical Sciences and Mathematics Commons

BYU ScholarsArchive Citation
https://scholarsarchive.byu.edu/etd/9662

This Thesis is brought to you for free and open access by BYU ScholarsArchive. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of BYU ScholarsArchive. For more information, please contact ellen_amatangelo@byu.edu.
Exploring Non-Functional Coupling Between Subsystems

Matt Pope

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of
Master of Science

Jonathan Sillito, Chair
Eric Mercer
Daniel Zappala

Department of Computer Science
Brigham Young University

Copyright © 2021 Matt Pope
All Rights Reserved
ABSTRACT

Exploring Non-Functional Coupling Between Subsystems

Matt Pope
Department of Computer Science, BYU
Master of Science

Many software systems depend on other systems to function correctly or are themselves composed of interdependent subsystems. In that context, behavioral changes in a dependency may have consequences for a dependent subsystem. This includes changes to non-functional properties such as latency or availability. In this paper we use the term *non-functional coupling* to mean the extent to which a subsystem is affected by changes to non-functional properties in a dependency. We argue that non-functional coupling has implications for the maintainability, reliability and performance of an overall system. We also explore the extent to which various engineering techniques used in interaction code (i.e., the code in a subsystem that manages requests to and responses from a dependency) can influence the strength of that coupling. We do this by simulating various techniques (including several novel techniques) using a tool named Quartermaster.

Keywords: software coupling, interaction, non-functional property
ACKNOWLEDGMENTS

Thanks go my advisor for my advisor, Dr. Jonathan Sillito and the members of the software engineering lab.
Table of Contents

1 In preparation: Exploring Non-Functional Coupling Between Subsystems  1
Chapter 1

In preparation: Exploring Non-Functional Coupling Between Subsystems

This manuscript has not yet been accepted for publication.
Exploring Non-Functional Coupling Between Subsystems

Abstract—Many software systems depend on other systems to function correctly or are themselves composed of interdependent subsystems. In that context, behavioral changes in a dependency may have consequences for a dependent subsystem. This includes changes to non-functional properties such as latency or availability. In this paper we use the term non-functional coupling to mean the extent to which a subsystem is affected by changes to non-functional properties in a dependency. We argue that non-functional coupling has implications for the maintainability, reliability and performance of an overall system. We also explore the extent to which various engineering techniques used in interaction code (i.e., the code in a subsystem that manages requests to and responses from a dependency) can influence the strength of that coupling. We do this by simulating various techniques (including several novel techniques) using a tool named Quartermaster.

I. INTRODUCTION

Many software systems depend on other systems to function correctly or are themselves composed of interdependent subsystems. In this paper we say that one subsystem depends on another subsystem when some functionality requires it to make a request to and get a response from that dependency over some network protocol. This definition does not cover all possible types of dependencies, but it provides an interesting subset for this paper. In general, a subsystem may have multiple dependencies and may itself be a dependency for multiple other subsystems (called its clients or dependents). Where dependencies exist, it is possible to consider the coupling between a pair of subsystems, which is a measure of the extent to which changes in one subsystem can affect the other. In this paper we are particularly interested in the effect of changes related to found types of non-functional properties: load, capacity, latency and availability. Each of these properties are defined in Section II.

For subsystems that have dependencies, part of the subsystem’s implementation is responsible for managing requests to and responses from its dependencies, and we refer to that as its interaction code. How that interaction code in each subsystem is engineered significantly impacts the reliability and performance of the overall system, and also impacts the ongoing effort required to maintain that reliability and performance. A naive implementation of a subsystem’s interaction code will mean that if any of its dependencies degrade, it will degrade as well and its dependents will be transitively affected. Similarly, if the rate of requests from a dependent increases, a subsystem will pass on that increase to its dependencies, possibly beyond the number of requests that can be supported, which may lead to an outage or degradation. In general, interaction code that is only engineered to effectively account for the “normal” behavior (say behavior specified in a service level agreement), will be brittle with regard to exceptional situations and ongoing changes in its dependents and dependencies.

In practice, interaction code tends to use various fault-tolerance techniques to deal with various exceptional scenarios. The goal of such techniques is to minimize the extent to which failure modes in a dependency can affect its dependents, possibly allowing a subsystem to gracefully degrade, and also on the flip side, to minimize the ability of a dependent subsystem to adversely affect its dependencies. Example techniques include load-shedding, setting timeout values for connections, employing various retry strategies, using the circuit breaker design pattern, caching responses from the dependency. Naturally, the way these techniques are used depends on the application context and on the design goals of the overall system.

However, it is difficult to anticipate and plan for all possible failure scenarios, and even the “normal” behavior of a dependency may change over time, and simply considering a dependency to be in one of two states (normal vs. failed, say) does not appropriately account for the range of possible changes to non-functional properties. These changes might be intentional or accidental, temporary or permanent, improvements or regressions. Regardless of the nature of the change, ideally a subsystem’s interaction code is adaptable enough to deal with these changes, implying a low degree of non-functional coupling.

In this paper we argue that non-functional coupling has implications for the maintainability, reliability, and performance of an overall system. We initially make this argument in Section III, where we discuss the nature of non-functional changes and the associated coupling. We then explore the extent to which various engineering techniques used in interaction code can influence the strength of that coupling, by considering existing techniques and four novel techniques (see Section IV). We conduct these explorations using a simulation tool named Quartermaster that is described in Section II. The novel techniques present ideas for increasing the ability of a subsystem to adapt to changes. Looking beyond the interaction code of an individual subsystem, we conclude the paper with a discussion of three higher-level techniques that we argue can further manage coupling between subsystems and reduce manual system maintenance due to changes in non-functional properties of its subsystems (see Section V).
II. BACKGROUND AND RELATED WORK

A. Coupling

Coupling can be defined as “the degree of interdependence among the components of a software system” [3] or “the measure of the strength of association established by” connections between modules [32]. Coupling has been shown to have consequences for various software engineering activities (e.g., [22], [36]) and for software maintainability and reliability generally [4], [18]. Intuitively, highly interrelated software components are harder to understand, change or correct.

Various types or sources of coupling have been identified or proposed in the relevant literature (e.g., [3], [21]). Fregnan et al., identify 22 coupling relations and group them into four categories: structural, dynamic, semantic and logical [8]. For instance, common coupling (which occurs when multiple components have access to the same global data [18]) is an example of structural coupling. Various metrics for measuring the degree of coupling between software components, as a tool for engineers to analyze their systems, have been proposed (e.g., [3]). Computing these metrics can involve static code analysis, dynamic analysis techniques, semantic analysis of various artifacts, and change histories (e.g., [9], [12]).

Coupling between subsystems has not been as widely discussed in the literature, and many of the types of coupling identified are not generally applicable to subsystems (though see Qian, et al.’s work on coupling metrics for services composition [27]). Further, we are not aware of coupling metrics that attempt to measure coupling due to non-functional properties of a subsystem.

B. Non-functional Properties

Non-functional properties “represent the description of the service characteristics that are not directly related to the functionality provided” [6]. In this research we are particularly interested in four non-functional properties of a system or subsystem, each of which are defined below.

The load from a subsystem is the number of requests sent to a dependency in a time interval. The load on a subsystem is the number of requests arriving from its dependents. Capacity is the maximum load a subsystem can service. When load exceeds capacity, a component may reject (i.e., immediately fail) excessive requests (a technique called load-shedding) or it may become overloaded. Availability is “the degree to which a system or component is operational and accessible when required for use” [11], and may be measured as \( R_s / R_t \), where \( R_s \) is the number of successful responses and \( R_t \) is the number of total responses over a time interval. Latency or response time is the time interval between the instant a request is made to a subsystem and the instant at which the response data has returned fully. The latency of a subsystem is often modeled as a distribution of the subsystem’s response latencies.

C. Interaction Code Techniques and Components

Software architecture “is concerned with the selection of architectural elements, their interactions, and the constraints on those elements and their interactions” [25]. A connector is an architectural element that models the interaction among two subsystems and those constraints [30]. The conceptual connector, which is a first-class entity and describes many facets of subsystem interaction, often differs from implementation, which may not have dedicated code and may be distributed in many places [23]. The aspect of connector implementations that we are interested in and discuss in this paper is the interaction code, which we limit to include only the code that makes decisions about the actual interactions (requests and responses) between subsystems.

In addition to tuning interaction code based on the normal non-functional behavior of its dependencies and dependents, it is also common to use various fault-tolerance [13], [28] techniques in anticipation of possible failure scenarios. Time-out and retry strategies can handle transient failures by performing additional interactions with a dependency. Caching and failback behaviors are forms of compensating actions, in which some level of service can be obtained from another source besides the dependency. Rejecting requests when load is nearing or exceeding capacity of a subsystem is called back pressure [35]. Doing so when some predetermined load is reached is called rate limiting. Similarly, load-leveling and load-shedding help smooth out intermittent loads by deferring some requests to a later time or preemptively failing some requests [2], [19]. Finally, the circuit breaker technique prevents the dependency from being called when it is likely to generate a failure [20].

The preceding list of techniques is incomplete, however based on our experience, it does cover most of the most widely used techniques.\(^1\) In Section IV we will discuss these techniques further, expanding on them in various ways. Each of these techniques can be said to play a decoupling role in the design of a subsystem’s interaction code by preventing cascading failures and mitigating degradations. Adaptability captures the “extent to which a software system adapts to change in its environment” [33]. The best adaptation method (i.e., the appropriate set of techniques used in interaction code and their configurations) may change over time as subsystems’ non-functional properties evolve [24].

A subsystem makes use of various software components to implement the above techniques, as shown in part B of Figure 1. Three of the most commonly used components (and the components we will focus on in this paper) are queues, pools, and caches. A request queue is an ordered sequence of pending requests to send to a dependency in which requests enter from one end of the sequence and exit from the other. A worker thread pool contains a number of threads, which when free, dequeue pending requests from the queue and call the dependency to obtain a response for the request. Once a request has obtained a response, the worker thread is free to service another request in the queue. Responses from a dependency may be stored in a request cache for a period of time called the time-to-live (TTL), for use in future requests.

\(^1\)Other popular engineering techniques exist that are less closely related to interaction code, and so are not considered in this paper (component level redundancy, autoscaling, load balancing, health checks, etc).
D. Quartermaster

In this research we have used Quartermaster, a modeling and simulation tool, to explore various techniques that could be used in a subsystem’s interaction code, with the goal of understanding how those techniques influence coupling between subsystems. Details about Quartermaster have been published previously [26] and it is publicly available,\(^2\) but for completeness we also describe it briefly here.

With Quartermaster, a user can create a model of a software system of interest, including the fault-tolerant techniques used in the system, by writing TypeScript code. Once a model of a system has been created, Quartermaster can simulate the execution of that model under various scenarios. The output of the simulated execution allows a user to understand how the model behaved in the given scenario. A scenario in Quartermaster is the contextual information used to drive the simulation, and it can change over the course of the simulation. At a minimum, defining a scenario involves specifying: (1) the rate of event arrival, and (2) parameters for the keyspace, which defines the distribution of unique requests.

III. NON-FUNCTIONAL COUPLING

In this paper we are discussing what we call non-functional coupling, which we define as the extent to which changes in the non-functional properties of a subsystem affect dependents or dependencies. In this section we argue that non-functional coupling (like other forms of coupling) has implications for the maintainability, reliability and performance of subsystems and the larger systems they comprise. In the next section we further explore the reliability and performance implications of non-functional coupling under various scenarios (see Section IV).

The four types of non-functional properties we are focusing on in this paper are load, capacity, latency and availability. In some contexts, some of these non-functional properties may be formally specified (in a service level agreement, say) and that specification may form the basis of engineering decisions made while implementing a dependent subsystem’s interaction code. When no formal specification is available, the “normal” behavior of a subsystem may be used in roughly the same way a formal specification is. Regardless of the level of formality in a specification, subsystems may be designed and configured in ways that encode assumptions about those non-functional properties and introduce non-functional coupling.

Dependents or dependencies may break those assumptions in at least two ways. First, a property may temporarily or intermittently diverge from its expected value (or expected distribution), possibly as part of a failure. Second, a property may be changed permanently, possibly as part of code deployment or other modification. In either case, any dependent subsystems that can not respond well to those changes (i.e., has not already been engineered to be sufficiently adaptive) may need to be modified to either appropriately tolerate the temporary variation that was experienced (and which may reoccur) or to accommodate the new normal. We next discuss these issues further and present illustrative examples from real incident reports with the goal of demonstrating the consequences of non-functional coupling.

A. Load, Capacity and Coupling

At any given time, a subsystem may be appropriately scaled to accommodate the load across all of its dependents. Though as circumstances change it may end up with excess or insufficient capacity. For example, if a subsystem is scaled by adding or removing virtual machines to a fleet, a hardware failure may reduce the total number of virtual machines in the fleet, temporarily reducing the capacity until new virtual machines can be added. It is also possible that load from a dependent may change over time, either temporarily (which we might call a “spike” in traffic) or permanently (say due to business growth). When load from one dependent increases, the available capacity for other dependents decreases, as long as total capacity remains unchanged. And such load changes can have cascading effects on downstream dependencies.

A subsystem that receives requests in excess of capacity may attempt to service the request, likely leading to degraded service for all requests (reducing performance of the system) and a complete outage if load is sufficiently high (reducing the reliability of the system). Various engineering techniques are used to tolerate such mismatches between load and capacity. Examples include load-shedding and load-leveling, and in Section IV we explore the extent to which these techniques reduce coupling associated with load and capacity. In Section V we discuss scaling approaches for adjusting capacity based on load.

A July 2015 incident in which an event processing system (CircleCI) failed, is an example of how changes in load from one subsystem can affect other subsystems (along with the overall health of the system).\(^3\) The parts of the system related to the failure include a queue that receives events from an external source, and a service that reads events from the queue and queries a database. The failure was precipitated by an unusually large number of incoming events to the queue, over a

---

\(^2\)https://github.com/BYU-SE/quartermaster

\(^3\)https://circleci.statuspage.io/incidents/hr0mm9xm3x6
short period of time. This surge in events led to a large number of concurrent database queries causing the database to become overloaded. Throughput dropped and the queue continued to grow until engineers mitigated the incident.

B. Latency and Coupling

The distribution of response times (or the latency) from a subsystem can be a source of coupling. In general, an increase in latency (mean latency, say) for a subsystem may increase the latency of any dependent subsystems. A latency sensitive subsystem may be configured with a timeout value on requests to a dependency, which represents the maximum time it is willing to wait for a response from that dependency, possibly mitigating increases in latency, at the cost of additional failure responses (due to timeouts). Like other non-functional properties, a subsystem’s latency distribution can temporarily or permanently change, and that change can be an improvement or a regression. For example, the deployment of a new version of a subsystem’s source code may introduce a latency change for some or all request types.

An October 2018 incident in which an online service (GitHub) experienced 24 hours of degraded service, is an example of coupling arising from assumptions about latency between subsystems.4 Due to a network interruption, a database cluster was automatically reconfigured, with the result that the primary database node was moved to a different datacenter. Applications that wrote to that database node were “unable to cope with the additional latency introduced by a cross-country round trip for the majority of their database calls.” Engineers mitigated the incident by moving the database node to the original datacenter to restore expected latency.

C. Availability and Coupling

In our experience many examples of reductions in availability, including the CircleCI incident described above, are due to load/capacity mismatches. And many times, timeouts of long running requests are considered failed responses (as in the GitHub example above). However, independent of load and latency, there are other reasons a subsystem may return failed responses and its overall availability may decline. Such failures may be intermittent, transient or permanent and can have different causes such as source code defects or hardware failures [1], [31]. Regardless of the nature or cause of the failure, availability of a subsystem can affect the availability of its dependent subsystems, though in some contexts this can be mitigated to some degree using caching techniques (or other fallback mechanisms) or retry strategies, which we explore in Section IV. However, that retries from a dependent subsystem may also contribute to further reducing a subsystem’s availability due to an increase in load.

A January 2018 failure is an example of a system (Elastic Cloud) architected on the (implicit) assumption of high availability.5 In this system, subsystems depend on a shared datastore and each subsystem instance stores a mirror of the


### Table I
A summary of the 12 Quartermaster models used in our simulations. In Column Comps (short for Components), Q = queue, P = worker thread pool, and C = response cache.

<table>
<thead>
<tr>
<th>#</th>
<th>Comps</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Naive: Per-request thread calls Z and responds to X when Z responds.</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>Q+P</td>
<td>Load-leveling: Requests are queued and serviced by threads in the worker pool.</td>
</tr>
<tr>
<td>C</td>
<td>Q+P</td>
<td>Load-shedding: Same as load leveling but queue is bounded and excess requests rejected.</td>
</tr>
<tr>
<td>D</td>
<td>Q+P</td>
<td>Multilevel load-shedding: Shed lowest value requests, with importance class specified by X in request header.</td>
</tr>
<tr>
<td>E</td>
<td>C</td>
<td>Response caching: Responses from Z are cached and used in place of a call to Z when present.</td>
</tr>
<tr>
<td>F</td>
<td>C</td>
<td>Async cache loading: Immediately return cached value (or failure); asynchronously load cache.</td>
</tr>
<tr>
<td>G</td>
<td>C</td>
<td>Per-request timeouts: Same as previous, but wait a per-request timeout supplied by X before cache read.</td>
</tr>
<tr>
<td>H</td>
<td></td>
<td>Retries: On failed responses from Z, Y retries call to Z up to a maximum number of tries.</td>
</tr>
<tr>
<td>I</td>
<td>C</td>
<td>Asynchronous retries: Building on model F, retry failed attempts to asynchronously load cache.</td>
</tr>
<tr>
<td>J</td>
<td>Q+P+C</td>
<td>Infinite retries: Building on I, use a priority queue and worker thread pool to control cost of retry strategy.</td>
</tr>
<tr>
<td>K</td>
<td>Q+P</td>
<td>Cooperative pool sizing: Dynamically adjust pool size based on importance class specified by X.</td>
</tr>
<tr>
<td>L</td>
<td>Q+P</td>
<td>Inferred pool sizing: Dynamically adjust pool size based on inferred capacity of Z.</td>
</tr>
</tbody>
</table>
we find that relative values (and the view they provide into overall behavior) to be more informative than absolute values and so the graphs below have no values on their axes.

Specifically, we have created Quartermaster models for 12 techniques and published them in a public repository. The models are summarized in Table I and described in detail in the following subsections. To the best of our knowledge, eight of these techniques are existing techniques and four are novel techniques we have introduced to further explore ideas for reducing coupling. For each of those models are assuming a simple system topology involving three subsystems (X, Y, and Z), where X depends on Y and Y depends on Z as shown in part A of Figure I. While simple, this structure is foundational for more sophisticated system topologies. And this topology allows us to consider coupling issues that arise due to possible behavioral changes in a dependent or a dependency.

Each of the four scenarios we have used for our Quartermaster simulations are characterized by a particular non-functional property that varies, as described below. Due to space limitations and other practical considerations, we have not simulated all potentially interesting scenarios and can not report on all the simulations we have run. Instead we have selected a small number of scenarios and report on a small number of metrics from the execution of those scenarios, that illustrate the key insights. Our focus is on the nature of the coupling that exists between systems, and how different techniques increase or reduce that coupling, along with the key tradeoffs that those techniques make.

A. Varying Load

In this Quartermaster scenario we use a sine function to vary the rate of requests sent to Y from X, simulating a pattern of increasing and decreasing load. Over the course of the scenario, the capacity of Z is kept fixed at a value selected so that the peak loads exceed Z’s capacity but the average load is manageable, and a request to Z in excess of its capacity results in a failure response. To explore coupling in the context of this varying load scenario we have created four models in Quartermaster that capture ways that the interaction code in Y might respond to that variation.

Model A: Naive Implementation. As discussed above, a naive implementation of a subsystem’s interaction code has one thread for each request it receives and it will call its dependency on that same thread which waits for a response. When the dependency responds, this same thread will respond to the subsystem’s caller. No effort is made to decouple the subsystem from dependents or dependencies, and it is included in this scenario largely as a point of comparison.

Model B: Load-leveling. Together, a queue and worker thread pool can provide an effect known as load-leveling [19]. When Y receives a request from X it stores that in the queue for future servicing. Threads in the worker pool read from the queue, send requests to Z and wait for a response. When Z responds, the worker thread responds to X and attempts to read another request from the queue. The size of the worker pool is an upper bound on the number of concurrent requests to Z, independent of the number of concurrent requests from X to Y. We selected the size of the pool to be a fixed value which would never overload Z.

Model C: Load-shedding. This technique known as load-shedding, can be added to the load-leveling technique by bounding the queue. If the queue is full at the time a request arrives from X, then it will be rejected and a failure response will be returned from Y without a corresponding call to Z.

Model D: Multilevel Load-shedding. In the case that not all requests are of equal value, a multilevel priority queue [29] (or similar) could be used in an attempt to ensure the highest value requests are processed in cases of excess load. For our purposes we assume that X supplies a priority class (one of high, medium or low) with each request, and the priority queue sorts requests by priority and when the queue is at capacity, it rejects the lowest priority request in the queue.

The results of these four simulations are summarized in Figure 2. Part A of the figure shows the extent to which the load from X is passed on to Z. As expected, model A (the naive approach) is directly coupled and the associated line on the graph can be used as a proxy for load from X. It is important to note that the portion of the model A line that is above the other lines represents requests that would result in failure responses from Z. The load-leveling model (B) and the two load-shedding models (C and D) which also load-level, never send more load than Z’s available capacity demonstrating the decoupling effect of the queue and especially the fixed sized worker thread pool. The differences between the model B, C and D lines on the graph reflect the requests that are shed by Y and therefore never sent to Z.

Part B of Figure 2 shows the consequences that the techniques from the four models have for the latency from Y to X, GitHub link will be in non-anonymized version.
Fig. 3. The results of simulating models A, E, G and F under a varying latency scenario. Part A shows the latency from Y to X as latency from Z varies. Part B shows the availability of Y to X as latency from Z varies.

and so gives some indication of the tradeoffs made when using a queue for decoupling load. The line for model A (which has no queue) reflects a latency that matches the latency from Z, while the other lines reflect additional queue wait time, where the wait time grows with the load. The load shedding from models C and D means that some requests do not wait in the queue but are rejected immediately. In model A the requests sent to Z in excess of its capacity do not have increased latency but result in failure responses. For this scenario, the mean availability from Y to X is: 91% for model A, 100% for model B, 94% for model C, and 94% for model D. For model D, all of the requests that were shed were low priority requests, and overall the multilevel load-shedding technique shows promise for decoupling from a dependent’s load, in application contexts where discriminating between requests is valuable.

B. Varying Latency

In this scenario we assume an application context in which X is latency sensitive and therefore uses a timeout value on calls to Y. We further assume that some requests from X may need quicker response times than others (see model G below for more). The latency (i.e., response time) from Z is modeled as a normal distribution and we use a sine function to vary the mean latency such that it ranges from 50% below to 50% above the timeout value set by X on its calls to Y.

To explore coupling as it relates to latency and techniques for reducing that coupling, we have created three Quartermaster models that use various caching techniques. As discussed above, these techniques are relevant in application contexts where it is possible to cache responses from a dependency for use in subsequent requests that are “identical”. In Quartermaster we use a key (which is sampled from a normal distribution) and requests with the same key are considered identical. Across all three models we have set the cache size to allow Y to store roughly two-thirds of the possible keys, and selected a TTL that will ensure responses in the cache expire up to eight times over the course of the simulation.

Model E: Request Caching. This model captures a simple caching approach in Y’s interaction code: when Y receives a request from X it will first check for a cached response and if found (called a cache hit) it will use that response rather than call Z. If not found, Y will call Z with a timeout value (corresponding to the timeout value set by X on its call to Y), and if a successful response is received before that timeout, it will load the cache and respond to X. If no response from Z is returned in time, it will not update the cache and will respond to X with a failure response.

Model F: Asynchronous Cache Loading. A different caching strategy has Y depending entirely on cached responses instead of waiting for responses from Z (called live responses), which allows very low response times from Y. In this approach, Y will immediately return a response to X based on whatever is currently in the cache. In the case of a cache miss, the response will be a failure, and in the case of a cache hit, the response will be a success. After Y responds it will always call Z in an attempt to load or refresh the cache for use in future requests from X with the same key.

Model G: Per-request Timeouts. For application contexts where some requests from X are more latency sensitive than others, we propose a technique in which X specifies a timeout value with its requests to Y. The technique builds on the asynchronous cache loading technique just discussed. The difference is that instead of immediately reading from the cache and responding, Y will wait for the specified amount of time before reading from the cache and responding. The delay introduces the possibility that the asynchronous cache load completes before the cache read occurs. In this model, we use three distinct and equally likely timeout values (fast, medium and slow) randomly assigned to request keys.

The results of these three simulations are summarized in Figure 3, along with the results of simulating model A in this same varying latency scenario. Part A of the figure shows the influence that Z’s latency has on the latency of Y. As expected, in model A the latency from Y mirrors the latency from Z (though it is truncated due to the timeout value imposed by X, as in all models). At the other extreme, models F and G (neither of which wait for a response from Z) exhibit constant latency. In the case of model F that latency is the time it takes to push to the queue and do a cache read. In the case of model G, the latency is equal to the timeout value supplied by X with the request, though on the graph we are only showing a mean computed for medium (rather than slow or fast) requests.

The latency from Y in model E is a mix of the two, as some responses are served from the cache and others (cache misses) call Z with a timeout, and the pattern is that latency from Y moves up and down with the latency from Z but the cache reduces the magnitude of the move (on average).

Part B of Figure 3 shows the consequences that the techniques have for the availability from Y to X, and so gives some indication of the tradeoffs made when depending on a cache.
The results of simulating models A, F, H, I and J under a varying availability scenario. Part A and B show the availability from Y to X over time. Part C shows the consequences for load on Z.

The rate of failed responses shown from model A and E are a reflection of timeouts and increase as Z’s latency increases. For models F and G, which do asynchronous cache loading, the availability is tied to the cache hit rate. The availability improvement of model G over F is due to the delayed read time that increases the probability of a cache hit. For this scenario, the mean availability from Y to X is: 42% for model A, 49% for model E, 45% for model F, and 71% for model G. Though if we break down the availability for model G based on the three types of requests, we see 68%, 71% and 75%. Overall the per-request timeout technique allows a subsystem to effectively use the available time budget, independent of a dependency’s latency.

C. Varying Availability

In this scenario we keep the rate of arrival from X static (and within Z’s capacity) and also keep Z’s response time distribution static. We model the probability of a successful response from Z using a sine function that ranges from 0 (where all requests fail) to 1 (where all requests succeed). Our simulation runs for two periods of the sine function.

To explore coupling issues that arise as availability varies, and techniques for reducing that coupling, we have created three additional Quartermaster models that use retry strategies. Two of the models also use a response cache and that cache is configured as described in Section IV-C. As a point of comparison, we have also simulated model A (naive) and model F (asynchronous cache loading) under this scenario.

Model H: Retries. This model builds on model A by adding a simple retry strategy: a request thread in Y may call Z up to three times in an attempt to get a successful response (i.e., performing up to two retries). Limiting the attempts to three is arbitrary, but roughly typical in our experience. Note that the choice of a retry limit often attempts to balance availability (more tries increases the likelihood of success), cost (more tries places more load on the dependency) and latency (fewer retries reduces Y’s latency).

Model I: Asynchronous Retries. For this model, we are proposing a retry strategy that builds on the asynchronous cache loading model (model F): a cache loading thread (that is running asynchronously) may call Z up to ten times in an attempt to get a successful response. The selection of ten as the maximum number of tries is arbitrary but is deliberately larger than the number used for model H (three) as it may be appropriate to use more retries because the cache loading is done asynchronously and X is not waiting on a response.

Model J: Infinite Retries. Both model H and model I use retry strategies with a fixed retry bound and the retries do not discriminate between requests. This model is an attempt to explore techniques that are more flexible in the number of calls to the dependency. Like in the asynchronous retries model, in this model Y does not wait for a response from Z before responding to X. However, a bounded multilevel priority queue and worker thread pool are used to control both the load on the dependency and to ensure that that capacity is used for the highest priority requests (with request priorities supplied by X, as in model D). The retry strategy then is that when a worker thread receives a failure response from Z it pushes the associated request back into the queue, but when the queue is full it will evict the lowest priority request. Note that unlike with a more standard retry strategy (see model H), this approach does not block the worker thread, and the time spent in the queue encompasses a dynamic backoff strategy (as requests are added to the end of the queue) [14].

The results of five simulations (models A, F, H, I and J) are summarized in Figure 4. Parts A and B of the figure show the influence that Z’s availability can have on Y’s availability for the different techniques used in the simulations. Naturally, in model A the two values are directly coupled. Similarly, when using model F (which uses the asynchronous cache loading technique) the probability of a cache hit (and therefore the availability of Y) is heavily influenced by Z’s availability, as attempted cache refreshes fail. However, the retry strategies provide a measure of decoupling because the probability of a success over multiple tries goes up, though in the case of models I and J the retry will just increase the probability of a cache load for future use. The availability of model I (which made up to ten attempts to load the cache) stayed relatively level even through periods of low availability in Z. Through those same periods, model J’s availability declined (though less than for model F, which used no retries), and it is...
important to note that the availability of high priority requests remained high. If we break down the availability for model J based on the three priorities of requests, high, medium, and low, we see 57%, 47% and 40% respectively.

Part C of Figure 4 shows the consequences these retry techniques have for the load on Z, which can be seen as the cost of the technique with that cost increasing as Z’s availability decreases. For the retry techniques in models H and I, the cost is proportional to the number of retries attempted, which is capped at three and ten respectively. For model J, this was controlled by the size of the worker thread pool used by Y, which we set to be less than the capacity of Z, and we found that it effectively used the available capacity for the most important requests, while still providing some decoupling from a dependency’s declining availability.

D. Varying Capacity

In this final scenario we use a sine function to vary the number of requests that Z can concurrently service, simulating a pattern of increasing and decreasing capacity. The rate of requests from X is fixed at a value that is larger than Z can service even at peak, and (as in previous scenarios) requests to Z in excess of capacity result in a failure response.

To explore coupling in the context of varying capacity we have considered two models each with a different approach to adapting a subsystem’s worker thread pool size. When a worker thread pool is used in a subsystem’s interaction code, the size of that pool will dictate the maximum number of concurrent requests it will send to a dependency, and in general will be configured based on the expected capacity available in that dependency, but it may be desirable to modify the pool size as available capacity varies.

Model K: Cooperative Pool Sizing. This model assumes that changes in capacity are made known to Y explicitly. Though other approaches may be possible, in our model Z uses a response header to inform Y of its capacity, and Y resizes its worker thread pool accordingly. More sophisticated approaches to cooperating around load and capacity are discussed in Section V. This technique may be analogous to rate limiting or throttling implementations which inform dependents when service (capacity) will be available.

Model L: Inferred Pool Sizing. In this model, we propose a novel technique for inferring changes in capacity and adjusting the worker thread pool size accordingly, and can be seen as a generalization of the circuit breaker pattern discussed above. This inference technique makes use of the ratio of timely successful responses to total requests made. A low ratio results in a decrease in the number of workers (in a scaling fashion) and a ratio near or above one results in an increase in the number of workers (in a linear fashion). This technique relies on three configuration parameters, which describe (1) the rate at which workers are removed in a scaling fashion, (2) the rate at which workers can be added in the linear fashion, and (3) the time interval over which the ratio is computed.

The results of these two simulations are summarized in Figure 5. While not shown in the graph, model A in this scenario sends a constant load from Y to Z, but even a model that uses a worker thread pool but does not adapt the size of the pool will send excess traffic to Z as its capacity declines, and will not make use of the additional capacity if it increases. On the other hand, model K’s behavior shown in the graph demonstrates the potential of adaptive approaches to make optimal use of available capacity, though in many contexts it may not be possible for Z to communicate its capacity. Model L, where a heuristic is used to adapt pool size to Z’s inferred capacity, performs somewhere between the models A and K. As is shown in the figure, the updates to the pool size trail the actual capacity changes. One final note on the results of this simulation, the portion of requests that are sent to Z in excess of its capacity (and therefore elicits a failure response) is 19% for model A, 9% for model L and of course model K does not send excess requests to Z if responses are timely.

V. REDUCING MANUAL SYSTEM MAINTENANCE

A key lesson from the techniques we discuss above is that static techniques can be a source of coupling and are less effective when a subsystem needs to adapt to an evolving environment. In the following we discuss three ways to take this key lesson further, going beyond what would be considered the responsibility of a single subsystem’s interaction code. First, we consider how we might adapt configuration values holistically, aiming for a global optimum, rather than adapting values individually. Second, we discuss techniques for automatically and cooperatively adjusting capacity. Finally, we consider approaches to allocating capacity when load exceeds capacity and scaling limits have been reached. We have not yet explored these ideas in Quartermaster, but we present them here as possible future work for us or others and also to further illustrate decoupling issues and propose possible solutions.

A. Tuning and Optimization

In our discussion of techniques for reducing coupling to this point in the paper, we have considered non-functional properties (load, capacity, latency and availability) as separate sources of coupling driving separate engineering decisions. Across a system these decisions or the decision-making logic are encoded in multiple subsystem’s interaction code and its configuration. The specifics will vary, but examples from
Section IV include at least the following decisions: (1) worker thread pool size, (2) queue length, (3) response cache size and TTL, (4) timeout values on multiple operations, and (5) maximum retry counts.

The space of possible configurations for even a single subsystem is large and often must be tuned to a particular context and so represents a challenging engineering task. When aspects of the context change (including changes to dependents and dependencies) a reconfiguration or other modification may be needed. While it is possible to make these decisions (and set the associated configuration parameters) in isolation from each other, in an attempt to improve a particular measure (e.g., response time), without considering the overall effectiveness of the interaction code or the larger system, it may be less effective.

A possible solution to these challenges may be to treat the configuration of interaction code holistically as an optimization problem, with the solution space defined by the legal values of each configurable parameter. The optimization could involve multiple subsystems or just one and would need to be guided by an application specific objective function. If such an approach proved effective, we would potentially be removing a major source of coupling between subsystems—the hard coded configuration parameters. Also, if the application context changes, only the objective function would need to be adjusted, supporting a relatively higher-level and declarative style of programming.

B. Cooperative Autoscaling

In Section IV we considered three techniques for handling load from a dependent subsystem that is greater than a dependency’s capacity (load-leveling, load-shedding and smart load-shedding). A common alternative is to scale up the dependency as load increases (e.g., [15]). For example, if a subsystem is implemented as a load-balanced fleet of servers, it may be configured to add or remove servers from the fleet as CPU utilization on the servers increases or decreases. Similarly, some cloud datastores can be configured to automatically adjust provisioned capacity as load changes.

If load increases on one subsystem and it is (auto)scaled up to handle that load, the load may increase on any dependencies of that newly scaled up subsystem, and so those dependences may need to be scaled up as well. If the dependency can not be scaled up sufficiently, we are simply moving the load-capacity mismatch from one subsystem to another, demonstrating a type of coupling. So we argue that it is important to look at autoscaling (and scaling generally) as a negotiation between a subsystem and its dependencies. And if those dependencies have their own dependencies, this cooperative autoscaling idea should be viewed as a recursive process. We theorize that this approach would be helpful for reactive autoscaling (i.e., in response to changes in load) but also for planned scaling (due to seasonal changes, say) that is often done manually today and may involve negotiation between subsystem owners, rather than being handled automatically between the subsystems themselves.

Promise theory may be a useful way to model and implement cooperative autoscaling. Promise theory is a model of voluntary cooperation between actors, with intentions published in the form of promises [5]. In this context, a promise would be from a dependency to a dependent and would verify an amount of available capacity (available to that dependent, not a total capacity available to all dependents) for a specified period of time. With such a promise in hand, a dependency can make scaling decisions and publish similar promises to its dependent subsystems. Load that is in excess of promised capacity has the potential to be rejected, though in some scenarios it may be able to be serviced (say if other dependents are not consuming all available capacity).

C. Scaling Limits and Auction Models

Scaling up a subsystem in response to excessive load may be technically impossible (e.g., the subsystem’s architecture has hit a scaling limit) or prohibitively expensive. In other cases, scaling up may be undesirable because the excess load is believed to be temporary or occasional. In such cases, there can be periods of higher demand (load) across some set of dependent subsystems than supply (capacity) from a shared dependency giving rise to a resource allocation problem: to which dependent subsystems (and which requests from those subsystems) to allocate scarce capacity.

One possibility is that the shared dependency could allocate a portion of its capacity to each of its dependent subsystems. Each of those dependents can then adapt its worker thread pool size appropriately (as in model K) to ensure it does not exceed its allocated capacity and use smart load-shedding (as in model D) to ensure the available capacity is used for the highest value requests. However, while such an approach may be locally optimal, some requests shed by one dependent subsystem may be more valuable than requests serviced by a peer. Another problem is that there can be periods of times when one dependent is not utilizing its allocated capacity and its peer dependents are unnecessarily shedding load.

Several possible improvements involve a priority queue managed by the shared dependency. Rather than require dependent subsystems to make prioritization decisions locally, they can send all requests to the dependency, which can then make those decisions globally. Such a scheme requires a notion of relative value between requests and how that is done may depend on the (business) relationship between the subsystem owners. We argue that each dependent is best positioned to know the value of its requests, however allowing a dependent to specify the priority value for each request introduces an incentive to always specifying the maximum possible priority, preventing the dependency from making meaningful prioritization decisions. This possibility of abuse can be addressed by introducing a pricing mechanism, where priorities can be seen as “bids” in an auction and winning bids result in corresponding charges to a dependent subsystem’s owner. Auction models have been well studied [7] and may serve as a mechanism for adapting to changing load and capacity.
VI. MEASUREMENT AND METRICS

In this work we are not proposing metrics for measuring non-functional coupling between subsystems. However, we do want to discuss what an effective metric might look like and how it might be computed. It would ideally capture the extent to which a change in a non-functional property in a subsystem would affect dependents and dependencies, and in future work we plan to explore how automated computation of these metrics could be incorporated into a development or deployment workflow, with the goal of ensuring that non-functional coupling regressions are not inadvertently introduced into a system.

We expect that a coupling metric is not easily computable through examination of source code (and of course there are scenarios where the source code of a dependent or dependency is not available to the owners of a subsystem) but instead will need to be computed based on a set of runtime metrics, similarly to what we have done in Section IV with each model. In our case, Quartermaster provided the ability to simulate these scenarios so we could generate useful time series data from normal and exceptional scenarios. However, in the case of real systems running tests using mock objects [16], [34] or fault injection [10] frameworks, may be one way to generate these metrics. In other circumstances, it may be possible to use runtime application logs, generated during normal execution and during failure scenarios that occur naturally, though the availability of such data from a sufficiently wide range of exceptional scenarios may be limited.

Based on that time series data, various statistical techniques could be used to measure the degree to which the variance of one non-functional property contributes to a non-functional property in another subsystem. One example is uncertainty analysis [17]. An initial, simplistic measurement of coupling can be generated by computing the first-order sensitivity index, to measure the contribution of a non-functional property of a subsystem to some non-functional property of another. Such an approach would answer questions such as, for a 1% latency increase from a dependency, what change do we see in the dependent’s latency. However, first-order methods only analyze the effects of varying a single non-functional property. Other orders of sensitivity analysis (such as second order analysis) could improve this method, capturing (for example) the effect that increasing the latency of a dependency, along with the resulting reduction in capacity, has on a dependent.

VII. LIMITATIONS

The techniques discussed in Section V have not been implemented, specified in detail or explored in Quartermaster simulations. As a result, the conclusions we can draw are limited and from that perspective these techniques can best be thought of as possible directions for future work on decoupling. For the twelve techniques that we have modeled in Quartermaster (see Section IV) and so have been able to simulate in various scenarios, corroborating our results using real systems and more realistic contexts remains important future work.

Many of the twelve techniques (including several of the novel techniques in particular) that we have explored in Quartermaster, are likely to be applicable in only narrow system contexts, and so our results should not be over generalized. For example, in model L the heuristics used to infer the capacity of a dependency make certain assumptions about how the dependency behaves when load exceeds capacity, and these assumptions will not hold universally.

Our work has largely simulated the various techniques in isolation, under scenarios in which one non-functional property varies at a time (e.g., a dependency’s latency), and with a simple system topology. Therefore, we are ignoring what are likely important interaction effects between techniques and also insights from more complicated scenarios. Finally, our Quartermaster models represent simplified subsystems, excluding important properties and effects of systems, such as non-collocated subsystems and CPU task scheduling.

VIII. SUMMARY

Non-functional coupling has implications for the maintainability, reliability, and performance of systems. Changes to non-functional properties of a dependent or a dependency may require engineers to retune a subsystem to account for a previously unanticipated scenario. This may happen many times as a system evolves. Non-functional coupling can allow for changes in non-functional properties to cross subsystem interfaces and manifest in dependents and dependencies (a cascading failure is an extreme example of this). In this paper we have explored and demonstrated how interaction code implementation and design choices influence how much non-functional coupling there is between subsystems. For example, well chosen, but fixed configuration values (e.g., time-out values) will not perform well across a range of scenarios and may be a source of ongoing maintenance.

The ideal techniques allow a subsystem to work well with its dependents and dependencies, and also to adapt well to changes, so that it can continue to perform well with no additional maintenance. Novel examples of such techniques introduced in Section IV include load-shedding that is sensitive to differences in request priorities, per-request timeout strategies that decouple a subsystem from the latency of a dependency, a retry strategy that balances availability and cost, and an inferential approach to adapting to changing capacity in a dependency. These techniques are not applicable in all system contexts, but they demonstrate possibilities for adaptable interaction code, and as a result lower coupling between subsystems.

We have also introduced ideas for adaptable architectures, using techniques that go beyond a single subsystem and its interaction code. These include global optimization of configuration parameters, cooperative autoscaling, and auction models for situations where resources are scarce, and an appropriate billing model is possible. At a high level, these techniques point to flexible ways to make use of available resources to (optimally) satisfy system objectives, with limited hard coding of assumptions.
REFERENCES


