Journal of Cloud Computing: Advances, Systems and Applications

SpringerOpen

Context Caches in the Clouds

Journal of Cloud Computing: Advances, Systems and Applications 2012, **1**:7 doi:10.1186/2192-113X-1-7

Saad L Kiani (saad2.liaquat@uwe.ac.uk)
Ashiq Anjum (a.anjum@derby.ac.uk)
Nick Antonopoulos (n.antonopoulos@derby.ac.uk)
Kamran Munir (kamran2.munir@uwe.ac.uk)
Richard McClatchey (ichard.mcclatchey@uwe.ac.uk)

ISSN 2192-113X

Article type Research

Submission date 1 December 2011

Acceptance date 22 February 2012

Publication date 9 July 2012

Article URL http://www.journalofcloudcomputing.com/content/1/1/7

Context caches in the Clouds

Saad Liaquat Kiani[1*]
[*]Corresponding author
Email: saad2.liaquat@uwe.ac.uk

Ashiq Anjum[2]
Email: a.anjum@derby.ac.uk

Nick Antonopoulos[2]
Email: n.antonopoulos@derby.ac.uk

Kamran Munir[1]
Email: kamran2.munir@uwe.ac.uk

Richard McClatchey[1]
Email: richard.mcclatchey@uwe.ac.uk

[1]Faculty of Engineering and Technology, University of the West of England, Bristol, UK

[2]School of Computing and Mathematics, University of Derby, Derby, UK

Abstract

In context-aware systems, the contextual information about human and computing situations has a strong temporal aspect i.e. it remains valid for a period of time. This temporal property can be exploited in caching mechanisms that aim to exploit such locality of reference. However, different types of contextual information have varying temporal validity durations and a varied spectrum of access frequencies as well. Such variation affects the suitability of a single caching strategy and an ideal caching mechanism should utilize dynamic strategies based on the type of context data, quality of service heuristics and access patterns and frequencies of context consuming applications. This paper presents an investigation into the utility of various context-caching strategies and proposes a novel bipartite caching mechanism in a Cloud-based context provisioning system. The results demonstrate the relative benefits of different caching strategies under varying context usage scenarios. The utility of the bipartite context caching mechanism is established both through simulation and deployment in a Cloud platform.

Introduction

Context in computing terms is the information related to the users of computing systems, which includes their personal situations, digital and physical environmental characteristics.

Context-aware systems facilitate the acquisition, representation, aggregation and distribution of this contextual information in ubiquitous environments. Established context-aware systems predominantly utilize a broker or a context server to facilitate context provisioning from providers of context information to context consumers. Due to the distributed nature of sensors and services that provide raw data for context creation, and that of applications/services that utilize such data, the provisioning of contextual information is a non-trivial task.

Existing context-aware systems are mostly focused on small geographic and conceptual domains and the context provisioning function of these systems has not attracted in-depth attention. For instance, the temporal properties of contextual data are not utilized by existing context-aware systems to improve context provisioning performance through caching, grid and cloud based platforms. One of the key challenges in context-aware systems is the provisioning of contextual information about *anything, anytime and anywhere* [1]. Meeting this challenge requires an infrastructure that can reliably collect, aggregate and disseminate contextual information related to a very large user base over a large scale. Cloud computing is ideally placed to provide infrastructural support for meeting this challenge through its key characteristics of reliability, scalability, performance and cost effectiveness. However, context-aware systems have not yet taken advantage of this recent progress in the computing arena.

In addition to the intrinsic benefits of Cloud computing, contextual information itself has certain features that can aid in improving the performance of systems that deliver context information from context producing components to context consuming components. Context information remains temporally valid for a certain duration, which depends on the type of context data. This property of the context data can be exploited by employing context caches in context provisioning systems to improve the overall system performance as done routinely in distributed systems. Our motivation towards investigating this area builds on the observation that contextual data is central to the functional relevance of any context-aware system. With a significantly large number of users, devices, data sources and services involved in the end-to-end cycle of acquisition, reasoning, delivery and consumption of context information, inadequate infrastructure support in terms of storage, processing, and provisioning of contextual information can be the biggest hurdle in adoption of context-aware systems over a large scale. Caching is a well established performance improvement mechanism in distributed systems, and if employed in Cloud based context provisioning systems, can augment its infrastructure strengths and further improve the context provisioning function.

Context information is usually modeled using name-value pairs, software objects and structured or semi-structured records. Irrespective of the representation format, the context information has an ever-present temporal property i.e. the information remains valid for a certain period of time. For example, an instantiation of the *location* context of a user remains valid as long as the user remains in that location, the *weather* context of a user remains valid as long as the user remains within the geographic span whose weather information is quantified in a context instance, the *Wi-Fi* context of a device remains valid as long as the device is connected to a certain Wi-Fi hotspot. This temporal validity can be exploited in intermediate components of context-aware systems for improving the context-provisioning performance e.g. caching contextual data at a context broker can allow for the exploitation of

the *locality of reference* in order to reduce contextual query satisfaction time and reduction in the overall context related traffic in the system. While caching is an established mechanism for performance improvement in distributed systems, the pertinent issues have not been analyzed extensively in the domain of context-aware systems. Firstly, different types of contextual data have varying validity durations i.e. a certain scope of context information (location, activity, Wi-Fi, weather, etc.) may remain valid for a few seconds while another scope may remain valid for days e.g. user-profile, device settings and shopping preferences. Secondly, the access rate and patterns of context consuming applications (distribution of scopes in context queries) may vary according to the time of day, type of context consuming application and user activity. Since caches are practically limited in size, cache replacement policies have to be employed during the context query-response and the variances in scope distributions in the queries, rate of the queries and validity periods of context scopes greatly influence the effectiveness of the cache replacement policies. The comparative effectiveness of different cache replacement policies needs to be analyzed and empirically evaluated.

Mere analysis of the caching strategies for contextual data provisioning is insufficient in the absence of a platform where their benefits can be fully utilized. The evolving technological landscape, characterized by increasing technological capabilities of smart devices and their adoption by everyday users, the greater availability of digital information services and the emergence of smart environments with embedded digital artifacts point towards an emerging digital ecosystem where a significantly large number of users in inter-connected smart environments will be utilizing context-based services through different computational interfaces. The success of context-aware systems will depend on accommodating these emerging scenarios and meeting their wide-spectrum requirements will greatly influence their adoption. Specifically, these requirements include device and location independence during utilization of contextual services, reliability of the system infrastructure, scalability in terms of load, administration and geographic scale, and the performance of the overall system in terms of query-response times and quality of service. A cloud based context provisioning system will 1) allow access to context information through standardised and interoperable interfaces, which will facilitate device and location independence, and 2) provide reliability and scalability through *elastic* and redundant resources. However, simply enabling Cloud based provisioning will not utilize the temporal validity characteristic of the context data, which can exploit the *principle of locality* to improve query-response times and therefore positively influence the quality of service of the context-aware system as a whole. Keeping these expectations in view, this paper relates the delivery of the caching functionality through a Cloud based context provisioning system, but focuses primarily on establishing the suitability and relative effectiveness of different caching strategies for different types of contextual data. Once such effectiveness is established through experimental analysis, we analyse the performance of the caching strategies in a prototype Cloud-based context provisioning system.

We discuss related work in the following section and then describe the functional characteristics of our Context Provisioning Architecture. The experimental evaluation of the caching functionality, and that of cache replacement policies in context caches, is presented in the *Context cache* section. Based on the results of the experimental analysis, we propose a novel caching strategy for utilization in context provisioning systems and discuss its dynamic re-configuration based operation as well (*The bipartite context cache* section). After experimentally establishing the performance benefits of the novel caching strategy, we carry

out an evaluation of various cache replacement policies in a Cloud-based deployment of the Context Provisioning Architecture (*Cloud-based evaluation* section) . The paper is concluded in the *Conclusions and future work* section with a discussion of relevant points that chart the future direction of this work.

Related work

A number of server/broker-based context provisioning systems have been developed, e.g. CoBrA [2], SOCAM [3], JCAF [4], PACE [5], and MobiLife [6] but caching contextual information has not been targeted in these systems explicitly. The MobiLife architecture specifies context caching at the context provider component but this approach creates distributed context caches at each context provider, potentially saving computational load at the providers but not reducing the communication cost. The query from the context consumer has to traverse the complete round trip from the context provider via the context broker. This mechanism can be improved by building a collective cache based on the smaller caches at context provider level.

Buchholz et al. [7] discuss the importance of caching context in improving its quality. Ebling et al. [8] also highlight caching of context data as an important issue in designing context-aware services and this point is reiterated in [9]. Caching context requires that the validity of a context data instance can be specified. This can be achieved by the inclusion of temporal information in the context representation format. MobiLife is one of the few context-provisioning systems that specify a caching component at the architecture level. However, its context representation format [10] contains no metadata that specifies its temporal properties. A similar system is the Context Based Service Composition (CB-SeC) [11] that employs a Cache Engine for providing context based service composition. However, the CB-SeC system does not store context information but the whole context service in the cache. A Caching Service is demonstrated in the SCaLaDE middleware architecture [12] for use with Internet data applications. The focus of this Caching Service is on providing disconnected operation to mobile devices by keeping a mobile device cache consistent with a central cache in the network. However, no performance metrics are reported regarding the gains achieved by the use of this cache. Despite the established significance and usability of caching components in distributed systems, context aware systems have not, as yet, demonstrated their use. Some researchers have highlighted the importance of caching context information but no study has reported any results on the empirical gains of employing a context cache in a context provisioning system and this deficiency has served as the main motivation for our continuing study of this domain. The discussion presented in this paper builds on our earlier work that demonstrated one of the first empirical studies on caching contextual data in context provisioning systems [13].

The context provisioning architecture

The Context Provisioning Architecture is based on the producer (provider)-consumer model in which context related services take the roles of context providers or context consumers. These basic entities are interconnected by means of context brokers that provide routing, event management, query resolution and lookup services. The following paragraphs describe these three main components of the architecture. A Context Consumer (CxC) is a component (e.g. a context based application) that uses context data. A CxC can retrieve context information by

sending a subscription to the Context Broker (CxB) or a direct on-demand query and context information is delivered when and if it is available. The Context Provider (CxP) component provides contextual information. A CxP gathers data from a collection of sensors, network/cloud services or other relevant sources. A CxP may use various aggregation and reasoning mechanisms to infer context from raw sensor, network or other source data. A CxP provides context data only to a specific invocation or subscription and is usually specialized in a particular context domain (e.g. location). A Context Broker (CxB) is the main coordinating component of the architecture. Primarily the CxB has to facilitate context flow among all attached components, which it achieves by allowing CxCs to subscribe to or query context information and CxPs to deliver notifications or responses.

A depiction of the core system components described above is presented in Figure 1 emphasizing the complementary provision of synchronous and asynchronous context-related communication facilities. A number of useful applications have been developed based on this architecture. Further details of this architecture and industrial trials are described in [14], [15]. Context consumers and providers register with a broker by specifying its communication end point and the type of context they provide or require. This in turn enables a brokering function in which the context broker can look up a particular context provider that a context consumer may be interested in (e.g. based on the type of context being requested). The broker can cache recently produced context, in order to exploit the principle of locality of reference.

Figure 1 Broker based context provisioning. Basic broker based context provisioning component interaction

A distinguishing feature of this architecture is the federation of multiple context brokers to form an overlay network of brokers (Figure 2), which improves the scalability of the overall system and provides location transparency to the local clients (CxCs and CxPs) of each broker. This federation of context brokers is achieved with a coordination model that is based on routing of context queries/subscriptions and responses/notifications across distributed brokers, discovery and lookup functions and is described in detail in our earlier work [16]. This concept of context broker federation can be directly related to Cloud federation in which two or more geographically distinct or administratively independent Clouds cooperate in resource sharing and related functional operations, hence setting the conceptual foundation for federation of context-aware Clouds that exchange cross-domain context-information for serving their mobile/roaming users.

Context information is represented in the Context Provisioning Architecture using an XML based schema entitled ContextML. The defining principle in ContextML is that context data relates to an *entity* and is of a certain *scope*. The entity may be a user, a username, a SIP or an email address etc., and scope signifies the type of context data e.g. weather, location, activity and user preferences. Furthermore, a temporal validity is associated with ContextML encoded context data through the *timestamp* and *expiry* tags, which specify the time duration during which a specific context instance is considered valid. This feature of ContextML forms the basis of utilizing the caching function in the context brokers of the architecture.

The actual context information about a scope is encoded using named parameters, parameter arrays and complex parameter structures in ContextML elements. A parser, titled the ContextML Parser, has been implemented as a Java library for Java SE, EE and the Android

platforms that can be used by thee context producing and consuming applications for the processing of contextual information and other messages encoded in ContextML. The model of the contextual data-related elements and a discussion about various dimensions of ContextML is presented in an earlier work [17].

A single broker based prototype of the Context Provisioning Architecture has been deployed on a Cloud platform and work is under progress to enable a federation of many such Cloud-based instances to be federated together in order to exploit the scalability, reliability, performance and interoperability related benefits offered by the Cloud platform. Figure 3 shows a conceptual diagram of how the system components may operate in a federation of context brokers in the Cloud infrastructure for the delivery of contextual information to context consumers. Each context broker may be under the control of a different administrative authority but the federation between these context brokers (and semi-private Clouds) can allow the context consumers to utilize these brokers for acquiring contextual information. The federation features are beyond the scope of this paper and we will limit our focus to the specific feature of context caching in a single broker setup.

Figure 2 Consumer–Broker–Provider interaction. Simplified view of the federated broker based interaction.

Figure 3 Federated broker model. Architectural components of the Context Provisioning Architecture in the Cloud infrastructure

Context cache

Context consumers request context about a particular entity and scope by forwarding a ContextML encoded query to the context broker. The broker forwards the query to an appropriate context provider that can satisfy it. When the query-satisfying context information is available, the provider sends the context response to the broker. In the absence of a caching facility, the broker simply forwards the query to the querying consumer. The Context Provisioning Architecture utilizes a caching component that caches recently received contextual data in response to context queries, in addition to forwarding the response to the querying consumer. The context data remains in the cache for the validity period unless it is replaced by more recent context of the same scope/entity or has to be removed to free the cache due to cache size limits. The query processing and notification operations from the context broker's point of view are described in Algorithms 1 and 2 respectively.

Algorithm 1

Context broker query processing

WHERE $P = \{P_1, P_2, \ldots, P_3\}$, P is the set of all providers in the system

WHERE a query $Q = \{I_q, I_e, I_s, I_{CxC}, Q_p\}$ # I_q is the query ID, I_e is the entity ID, I_s is the scope ID, I_{CxC} is the consumer component's ID and Q_p consists of other query parameters.

WHERE $T_q = \{I_{q_i}, I_{CxC_i}\}$ # T_q is a table where the broker stores the query ID to consumer ID mappings of the form $\{I_q, I_{CxC}\}$

subscribe(Q) # Query arrives at the broker

$record(T_q, I_q, I_{CxC})$ # Query is recorded in the queries table

$CXT_f = searchCache(Q, I_e, I_s)$ # See if cache can satisfy the query

if CXT_f **then**

$notify(I_{CxC}, CXT_f)$ # Notify the consumer in case of cache hit

$incrementUseCount(CXT_f)$ # Increment the use count of the particular item

else

$P_s = lookup(P, I_e, I_s)$ # Broker looks up an appropriate provider

$query(Q, P)$ # And forwards the query to that provider

end if

Algorithm 2

Context broker notification processing

WHERE $T_q = \{I_{q_i}, I_{CxC_i}\}$ # T_q is a table where the broker stores the query ID to consumer ID mappings of the form $\{I_q, I_{CxC}\}$

WHERE T_{Ins} is the cached item insertion time

WHERE T_{Exp} is the cached item's validity expiry time

WHERE CXT_{i_n} is the cached item's use count

$publish(I_q, CXT_p)$ # Context response arrives from the provider

$storeInCache(CXT_p, T_{Exp}, T_{Ins})$ # Store the context item in the cache

$I_{CxC} = resolve(T_q, I_q)$ # Find out which consumer requested this context item

$notify(I_{CxC}, CXT_p)$ # Notify the consumer

In addition to the development and real-world deployment of the Context Provisioning Architecture system, a simulation model has been developed to evaluate the system under various conditions. The simulator is based on OMNET++ [18], a Discrete Event Simulator toolkit, and models the actual system components (providers, consumers, broker), the representation scheme and the communication model between these components as well. The results of the experiments carried out with this simulated setup will aid in establishing the suitability and relative effectiveness of caching strategies for context provisioning. These caching strategies can then be readily implemented in a Cloud-based Context Provisioning Architecture to augment the reliability, scalability and device/location independence benefits that are provide by the Cloud setup.

Simulation

The simulation model consists of a context broker module, context providers and context consumers connected by communication channels. The simulator comprises the core functionalities of context caching, context querying service, CxP registration and lookup service. Furthermore, the ContextML schema is also fully modeled. CxP modules provide context on invocation by the CxB and provide context about one particular scope only. The simulation model comprises various input parameters that can be set individually for each simulation run allowing several scenarios to be evaluated and compared against each other. The parameters for each scope contain numerical scope ID (integer) and its validity duration (seconds). The parameters for each Context Provider comprise a CxP ID (integer), ID of the context scope that it provides (integer) and the average time taken to process a query and respond to it (ms). The context broker module parameters include the lookup time for finding CxPs for satisfying queries (ms), cache access time [ms], caching enabled (Boolean), maximum cache size (integer i.e. the number of items in the cache), and the cache strategy i.e. the cache replacement policy used (integer).

$$scopeID = \left\lceil maxScopeID \cdot \left(\frac{\xi}{randUniform\,(0,1]} \right)^{-\sigma} \right\rceil \tag{1}$$

Within the scope of this evaluation, there are three main caching strategies that we will evaluate, including *remove oldest first (*OF*), remove least used first (LU), and remove soonest expiring first (SE)*, in addition to the non-practical strategy of having an infinite cache size thus requiring no replacement policy. When the cache is full and space is required for a more recent context item, the *OF* policy removes the oldest item from the cache. The context caching functionality in our system therefore records the time of insertion of each item in the cache. The *SE* policy removes the context item from the cache store whose *expiry* time will be up the soonest. In the case of *LU* policy, the context item which has been accessed the least number of times. For this policy to be applicable, the caching function in our system has to record each context item's access frequency. These cache replacement policies, with respect to their usage by the context broker, are described in Algorithm 3.

Algorithm 3

Context cache insertion and replacement procedure (`storeInCache`)

WHERE T_{Ins} is the cached item's insertion time

WHERE T_{Exp} is the cached item's validity expiry time

WHERE CXT_{i_n} is the cached item's use count

WHERE $policy = `LU' \vee `OF' \vee `SE'$ # The cache replacement policy

if $filledSpace < maxSpace$ **then**

$insert(\mathcal{MD}5(I_e||I_s), CXT, T_{Exp})$

else

if $policy == `SE'$ **then**

$CXT_{Rem} = Minimum(T_{Exp})$ # Select the item with the soonest reaching *expiry* time

$remove(CXT_{Rem})$

end if

else

if $policy == `OF'$ **then**

$CXT_{Rem} = Maximum(T_{Ins})$ # Select the item with the oldest insertion time

$remove(CXT_{Rem})$

end if

else

if $policy == `LU'$ **then**

$CXT_{Rem} = Minimum(CXT_{i_n})$ # Select the item with the least usage count

end if

end if

We have already established the usefulness of caching contextual data in principle in our earlier work [13], but did not analyze the effect of variance in the scope validity durations in detail. As the results will demonstrate, different access patterns from users (requesting longer validity scopes more than shorter validity scopes) can have a significant influence on the performance of the cache (cache-hit rate). With the help of this simulation model, we intend to establish suitable strategies for varying access patterns and devise a caching strategy that can accommodate a combination of these access patterns. The context consumers are configured to request context a constant rate λ [/s]. The context scope specified by the CxCs in the queries is determined using a Pareto distribution with a selectable shape α and scale ξ (1).

The discretized Pareto distribution has been selected because it allows us to model scope distribution in context queries with tuneable parameters. Twelve different scopes are used in this experiment and the scope distribution in context queries is controlled by changing the Pareto shape parameter α while the scale parameter ξ is kept constant at 1. In each simulation run 5,000 context requests distributed across 10 entities are instantiated. After all the responses have been received by CxCs, the simulation is terminated. Scopes and CxPs are initialized using the values from Table 1. The CxB cache access time and provider lookup time are assumed to be 10ms. In each simulation run, a caching strategy is selected and the Pareto distribution for selecting the requested scopes (α) is varied to select a certain percentage of *short validity (SV)* and *long validity (LV)* category scopes. The simulation is repeated for each caching strategy and the query satisfaction time, the time elapsed between issuance of a query from a consumer and receipt of a response to that query, being recorded. Hence, the performance of the selected cache strategies is investigated with varying scope distribution in the context requests.

Table 1 Simulation parameters

CxP:ScopeID	Processing time[ms]	Validity[s]	Category
CxP:1	70	60	Short
CxP:2	70	60	Short
CxP:3	80	80	Short
CxP:4	80	80	Short
CxP:5	90	180	Short
CxP:6	90	240	Short
CxP:7	70	360	Long
CxP:8	70	400	Long
CxP:9	80	600	Long
CxP:10	80	900	Long
CxP:11	90	1200	Long
CxP:12	90	1200	Long

Results

The mean query satisfaction times of 5000 context queries with different caching strategy are plotted in Figure 4 . We analyze the mean query satisfaction time of these caching strategies in the cases where scope distribution varies from being fully focused on short validity (SV) scopes to long validity (LV) scopes in increments of 25% i.e. the distributions range from (1.0 SV/0.0 LV), (0.75 SV/0.25 LV), (0.5 SV/0.5 LV), (0.25 SV/0.75 LV) and (0.0 SV/1.0 LV). The reference cases of having an unlimited and no cache show the maximum performance

improvement possible with our setup. The mean query satisfaction time across different combinations of SV/LV scope distributions improves from 487ms to 292.8ms, with a cache-hit ratio of approximately 46%. However, having an unlimited cache size is impractical in deployment scenarios, hence we focus our attention to various cache replacement policies that are evaluated with a fixed cache size of 500 items maximum i.e. 1/10th of the total number of context items that will be generated during an experimental iteration.

Figure 4 Simulation experiment results. Mean query satisfaction times, different caching strategies and scope distribution scenarios

The caching sub-component in the context broker keeps track of the number of times an element in the cache has found use i.e. cache-hits that have occurred. It also records the time of arrival of a context-item in the cache and time left in the expiry of a context data item's validity. When space is needed in the already full cache for a newer context data item, the LU cache replacement policy removes an existing item from the cache that has been accessed the least number of times. The chart in Figure 4 shows that LU results in mean query satisfaction time of 358.8ms (with ˜33.5% cache-hit ratio) and provides a fairly even performance for both the short validity scope and long validity scope focused context queries. The *OF* cache replacement policy provides an improvement over LU with a mean query-satisfaction time of 341.8ms. However, it is evident by considering the results in Figure 4 that *OF* delivers a better query satisfaction time when the scope distribution in the contextual queries is biased towards SV scopes. This can be explained by the fact that under a querying pattern where most of the queries contain requests for SV scopes, the SV context data items will dominate the cache store. But since these data items have shorter validity durations, by the time they are removed due to the *OF* policy they would be closer to the expiry instant and hence been offered a greater chance of generating a cache-hit by spending most of their validity period in the cache. In the reciprocal case of high concentration LV scopes in the context queries, *OF* policy results in the longer validity data items from the cache that are not often closer to their expiry instant and hence have not been offered a fuller chance to result in a cache-hit.

The *SE* cache replacement policy removes an item from the cache that is the closest to its validity expiration. This policy delivers an improved mean query satisfaction time of 331.4ms (with ˜36.25% cache-hit ratio) across all scope distributions but a closer inspection reveals that *SE* performs better for LV scoped queries than SV scoped queries. This policy is biased towards replacing an SV scoped item from the cache store because the validity expiry time for such items is more than likely to be closer than LV items. Moreover, an LV scoped removal candidate item would have spent most of its validity duration in the cache and thus given a good chance to result in a cache-hit.

The *OF* and *SE* policies can be further examined by comparing the validity categories of data items that resulted in a cache-hit. Figures 5 and 6 illustrate the relative percentage of SV and LV scoped items in the cache-hit resulting data items. It can be seen from Figure 5 that under the *OF* policy, SV scoped data items occupy a share of the context-hit space that is greater than their percentage in the context queries i.e. under the *OF* policy it takes a 78% share in the case of 0.75 SV/0.25 LV, 52% in case of 0.5 SV/0.5 LV and a 28% share in the case of 0.25 SV/0.75 LV. Contrastingly, under the *SE* policy, LV scoped data items occupy a greater share of the context-hit space i.e. under the *SE* policy it takes a 30% share in the case of 0.75 SV/0.25 LV, 58% in case of 0.5 SV/0.5 LV and a 80% share in the case of 0.25 SV/0.75 LV

(see Figure 6). These trends demonstrate the suitability of *OF* and *SE* policies for SV and LV scoped context data respectively. We have used these observations to devise a novel caching mechanism for contextual data that is suitable for both short and long validity scoped context data, which is discussed in the following section.

Figure 5 Cache-hit ratio under *OF* cache replacement policy. OF replacement policy and cache-hit rate of SV vs. LV in different scope distribution scenarios

Figure 6 Cache-hit ratio under *SE* cache replacement policy. SE replacement policy and cache-hit rate of SV vs. LV in different scope distribution scenarios

The bipartite context cache

Taking into consideration the suitability of different cache replacement policies for SV and LV scope categories, we split the physical cache into two parts, one catering for the SV scoped context data items and the other for LV scoped items. The caching strategy is then configured to utilize *OF* replacement policy for SV scoped data and *SE* policy for LV scoped data items. The performance of the bipartite context cache is evaluated under the same experimental conditions discussed earlier and the results are plotted in Figure 7. The bipartite cache provides a marginally improved overall performance over the *OF* and *SE*, with a mean query satisfaction time of 326ms (˜38.1% cache-hit ratio). This use of two different cache replacement policies suited to the scope validity durations of the data items results in an improved performance and provides a fairly constant mean query satisfaction time across all scope distribution patterns.

Figure 7 Simulation experiment results including the bipartite cache. Inclusion of the Bipartite with Dynamic Size cache strategy in the earlier comparison

We have further evaluated the bipartite caching mechanism with a dynamic scaling of the size of the two partitions that is based on the distribution of scopes in the incoming context queries. Dynamically increasing or decreasing the size of a partition based on the ratio of a particular scope validity category in the incoming queries tunes the cache to accommodate the pattern of queries that exists in a particular situation. The query satisfaction times improve marginally by the application of bipartite caching with dynamic partitioning from the case of equally sized bipartite cache. The mean query satisfaction time in our experiments is 318.4ms (˜39.4% cache hit-ratio) and the results display a consistent pattern across all scope validity scenarios (Figure 7).

The experiments carried out in a simulated environment (OMNET++) have established that the bipartite caching mechanism, both with fixed and dynamically resizable partitions, operating different replacement strategies in both partitions provides a better hit ratio. Hence, the mean query satisfaction time improves in comparison to the *SE* and *OF* policies operating independently. In the following section, we describe the deployment of our broker based Context Provisioning Architecture system in a Cloud platform and repetition of the previously carried out experiments.

Cloud-based evaluation

The prototype implementation of the Context Provisioning Architecture has been carried as a collection of applications and services conforming to the Enterprise JavaBeans [19] specification. The Context Broker EJB application consists of functional entities that provide the registration, brokering, querying, notification and caching functions through RESTful HTTP interfaces. The *Cache Service* is implemented as a Singleton Session Bean of the EJB specification with appropriate interfaces for adding, removing and retrieving context information. Singleton session beans are designed for circumstances in which a single enterprise bean instance is shared across and concurrently accessed by clients, which in this case are the constituent components of the Context Broker. The Cache Service has built in cache replacement policies (including those mentioned in the preceding discussion) and the administrator can configure which policy is to be used during execution. Generic Context Provider and Context Consumer applications have also been developed that can be programmed for querying and providing context information related to a particular scopes and entities at specified rates.

The ContextBroker is deployed on a Glassfish [20] application server, which itself is hosted on a compute node of the OpenStack (www.openstack.org) cloud platform (Diablo release). The OpenStack cloud platform is deployed on a server consisting of a *nova-compute* virtual machine. The software/hardware configurations of the server and the virtual machine, along with the physical and virtual network configuration details, are illustrated in Figure 8. The Context Providers are deployed on a server that is accessible to the Context Broker via a local area network through the host OS. The Context Consumers are deployed on a workstation on the public network available to the Cloud nodes. It must be noted that for the purposes of this experiment, the public interface is also confined to a local area network in order to reduce the variable network factors during execution. The local area networks consist of 1Gbps ethernet connections.

Figure 8 Cloud-based deployment. Deployment of the Context Provisioning Architecture on an OpenStack based Cloud for the caching experiment

As with the simulated experiments, the Cloud-based experiments are carried out with 5000 context queries, which are sent from context consumers to the broker. The mean query satisfaction times with different caching strategy are recorded for the cases where scope distribution varies from being fully focused on SV scopes to LV scopes. The results reported/plotted in the following paragraphs are a mean of five repetitions (the individual results being within +/–2% of the mean results). The results are plotted in Figure 9 and Figure 10. The chart in Figure 9 shows that *OF* cache replacement policy provides a better query satisfaction time when the context queries are focussed towards *SV* scopes while *SE* policy provides a better query satisfaction time when the query scope distribution is biased towards *LV* scopes. This results are in line with our earlier findings that are based on a simulation environment. However, an notable observation is the variation in the performance of a particular cache replacement policy as the scope distribution varies from one extreme to the other (i.e. 1.0SV/0.0LV to 0.0SV/1.0LV). In the case of simulation based experiments, the *SE* policy demonstrates a net 10.8% (315ms to 338ms) change in the query satisfaction times across the range of scope distributions while the *OF* policy demonstrates a similar 10.3% (321ms to 354ms) change as the scope distributions varies from 1.0SV/0.0LV to 0.0SV/1.0LV.

But when the experiments are carried out on the Cloud-based deployment, the query satisfaction times in case of *OF* and *SE* demonstrate a range that varies 14.96% and 17.5% respectively, i.e. the practical favourable tendency of these policies towards *SV* scoped queries (*OF* policy) and *LV* scoped queries (*SE* policy) is more pronounced.

Figure 9 Cloud-based experiment results. Bipartite cache with fixed partition size (equally distributed)

Figure 10 Cloud-based experiment results. Bipartite cache with dynamically resizable partitions

The performance of the bipartite caching mechanism demonstrates a trend that is similar to the earlier simulation experiments i.e. it provides an improved overall performance over the *OF* and *SE* policies applied independently, with a mean query satisfaction time of 263ms (˜40.2% cache-hit ratio). Moreover, the provision of a fairly constant mean query satisfaction time with the bipartite cache across all scope distribution patterns is also evident. The query satisfaction times in case of the bipartite cache with fixed partition sizes only vary by a 1.45% (262ms to 265ms) as the scope distribution varies between *SV* and *LV* scoped queries i.e. it provides a fairly consistent performance. Figure 10 illustrates the performance of the bipartite caching mechanism with dynamic partition resizing and the results confirm our earlier findings that signified the improved performance of this caching mechanism. The bipartite caching mechanism with dynamic partition resizing delivers a mean query satisfaction time of 245.8ms, which is better than 1) the *OF* results by 12.77%, (277.2ms), 2) the *SE* results by 7.89% (265.2ms) and 3) the bipartite cache using fixed partition sizes by 7.16% (263.4ms). Moreover, it demonstrates only a 0.82% change (from 245ms to 247ms) as the scope distributions vary from 1.0SV/0.0LV to 0.0SV/1.0LV, signifying a consistent performance during different query conditions. This consistency is in line with our earlier findings during the simulated experiments where the bipartite caching mechanism with fixed and dynamically resizable partitions demonstrated a deviation of 1.23% and 0.96% respectively as the scope distributions varied. The mean observed cache hit ratio in the case of the bipartite caching withe dynamically resizable cache is 41.36% (individual repetition of experiments yielded cache hit ratios of 41.2%, 41.7%, 40.9%, 41.1%, 41.9%).

The experiment carried out in the Cloud based deployment of the Context Provisioning Architecture have reaffirmed our findings from the OMNET++ simulation that the bipartite caching mechanism, both with fixed and dynamically resizable partitions, operating different replacement strategies in both partitions, provides a better mean query satisfaction time in comparison to the *SE* and *OF* policies.

Conclusions and future work

The work presented in this paper builds on the well established mechanism of caching in distributed systems for performance improvement purposes. However, the use and effectiveness of context caches has not been evaluated or demonstrated. The Context Provisioning Architecture employs a caching mechanism at the context broker, which positively affects the mean query satisfaction time between context consumers and providers. We have analysed the relative performance of various cache replacement policies using the OMNET++ discrete event simulator. Our analysis has revealed that different caching

strategies display contrasting behaviour under different scope distribution scenarios, with *OF* policy performing better for short scoped context data and *SE* performing better for long scoped context data. Based on this observation, we have devised a novel bipartite caching strategy for use in context data provisioning that allows utilization of the *OF* and *SE* policies for SV and LV scoped context data during context provisioning.

The bipartite cache is further improved by allowing dynamic resizing of the bipartite cache partitions based on the scope distribution scenario of the incoming context queries. The novel caching strategy can assist in designing a Cloud based context provisioning system that *effectively* utilizes the temporal validity characteristic of the context data, exploit the principle of locality to improve query-response times and therefore positively influence the quality of service of the context-aware system as a whole. To validate our claim, we have repeated the simulation based experiments on a deployment of the Context Provisioning Architecture in a Cloud platform. The subsequent results demonstrate similar trends and improvement in the mean query satisfaction times through the use of caching and the comparatively better performance of the novel bipartite caching mechanism.

The experiments presented in this work have been carefully designed to reduce the number of variable factors e.g. carrying out the experiments in isolated local area networks and configuring context providers with deterministic behaviour i.e. fixed processing and response delays. Such restrictions have assisted in determining the behaviour of the caching function to a better degree of confidence as the effects of variable factors is diminished in our setup. However, to derive more generalizable results, we aim to carry out the experiments under real-world conditions with the context providers and consumers deployed in public networks and consuming/providing a more practical context information set.

Furthermore, the Context Provisioning Architecture enables multiple brokers to be federated together in an overlay network and the geographically distributed context consumers and providers can be attached to different brokers. In such a setup, there will be multiple distributed caches in the system, one at each broker. The caching benefits cannot be maximised if such distributed caches are not synchronised amongst each other. Such a synchronisation mechanism does no exist in our system and is a target of our future work. The distributed, yet synchronized, caches may also effect the suitability of cache replacement policies to certain distributions of scopes in the context queries and we also expect that an investigation into this aspect may open further avenues of research and development in this domain.

Competing interests

The authors declare that they have no competing interests.

Authors' contributions

SLK drafted the manuscript, carried out the background work to identify the problem area and devised the caching strategy. AA participated in drafting the manuscript, designing the simulation parameters and carrying out the experiments. NA participated in the experiment design and analysis of the results with respect to Cloud-based feasibility of our approach. KM carried out the analysis of the experimental results and participated in the Cloud-based

evaluation. RM carried out the analysis of the experimental results and contributed significantly towards establishing the logical validity of our results. All authors have made substantial contributions and have been part of drafting the manuscript. All authors read and approved the final manuscript.

References

1. Weiser M (1991) The computer for the twenty-first century. Sci Am 265(3):94–104
2. Chen H (2004) An Intelligent Broker Architecture for Pervasive Context-Aware Systems. PhD thesis, University of Maryland, Baltimore County
3. Gu T, Pung HK, Zhang DQ (2005) A Service-oriented middleware for building context-aware Services. J Netw Comput Appl 28:1–18.
4. Bardram JE (2005) The Java Context Awareness Framework (JCAF) - a service infrastructure and programming framework for context-aware applications. In: Pervasive Computing, Volume 3468 of *LNCS*. Springer, pp 98–115
5. Henricksen K, Indulska J, McFadden T, Balasubramaniam S (2005) Middleware for distributed context-aware systems. In: Meersman R, Tari Z (eds) On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE, Volume 3760 of Lecture Notes in Computer Science. Springer, Berlin / Heidelberg, pp 846–863 http://dx.doi.org/10.1007/11575771_53.
6. Floreen P, Przybilski M, Nurmi P, Koolwaaij J, Tarlano A, Wagner M, Luther M, Bataille F, Boussard M, Mrohs B, et al. (2005) Towards a context management framework for mobiLife. 14th IST Mobile & Wireless Summit
7. Buchholz T, Küpper A, Schiffers M (2003) Quality of Context: What It Is and Why We Need It. In; Workshop of the HP OpenView University Association
8. Ebling M, Hunt GDH, Lei H (2001) Issues for Context Services for Pervasive Computing. In: Workshop on Middleware for Mobile Computing, Heidelberg. http://www.mobilesummit.de/authors.php
9. Lei H, Sow DM, Davis I, John S, Banavar G, Ebling MR (2002) The design and applications of a context services. ACM SIGMOBILE Mobile Comput Commun Rev 6(4):55
10. Kernchen R, Bonnefoy D, Battestini A, Mrohs B, Wagner M, Klemettinen M (2006) Context-awareness in mobiLife. In: Proceeings of the 15th IST Mobile Summit. IST Mobile Summit, Mykonos, Greece
11. Mostéfaoui SK, Tafat-Bouzid A, and Hirsbrunner B (2003) Using context information for service discovery and composition. In: Kotsis G, Bressan S, Catania B, Ibrahim IK (eds) 5th International Conference on Information Integration and Web-based Applications and Services (iiWAS). Östereichische Computer Gesellschaft, ISBN 3-85403-170-10
12. Bellavista P, Corradi A, Montanari R, Stefanelli C (2006) A mobile computing middleware for location and context-aware internet data services. ACM Trans Internet Technol (TOIT) 6(4):380.

13. Kiani SL, Knappmeyer M, Reetz E, Baker N (2010) Effect of Caching in a Broker based Context Provisioning System. In: Proceedings of The 5th European Conf. on Smart Sensing and Context, Vol 6446, LNCS. pp 108–121.

14. Zafar M, Baker N, Moltchanov B, João Miguel Goncalves SL, Knappmeyer M (2009) Context Management Architecture for Future Internet Services. In: ICT Mobile Summit 2009. Santander, Spain

15. Knappmeyer M, Tönjes R, Baker N (2009) Modular and extendible context provisioning for evolving mobile applications and services. In: 18th ICT Mobile Summit

16. Kiani SL, Knappmeyer M, Baker N, Moltchanov B (2010) A Federated Broker Architecture for Large Scale Context Dissemination. In: 2nd Int'l Symp. on Advanced Topics on Scalable Computing. Bradford, UK

17. Knappmeyer M, Kiani SL, Frá C, Moltchanov B, Baker N (2010) A Light-weight context representation and context management schema. In: Proceedings of IEEE International Symposium on Wireless Pervasive Computing

18. Varga A (2001) The OMNeT++ discrete event simulation systems. In: Proceedings of the European Simulation Multiconference (ESM'2001). pp 319–324.

19. Emmerich W, Kaveh N (2002) Component technologies: Java beans, COM, CORBA, RMI, EJB and the CORBA component model. In: Proceedings of the 24th International Conference on Software Engineering (ICSE 2002). IEEE, pp 691–692

20. Goncalves A (2009) Beginning Java EE 6 Platform with GlassFish 3: from novice to professional. From Novice to Professional Series, Apress

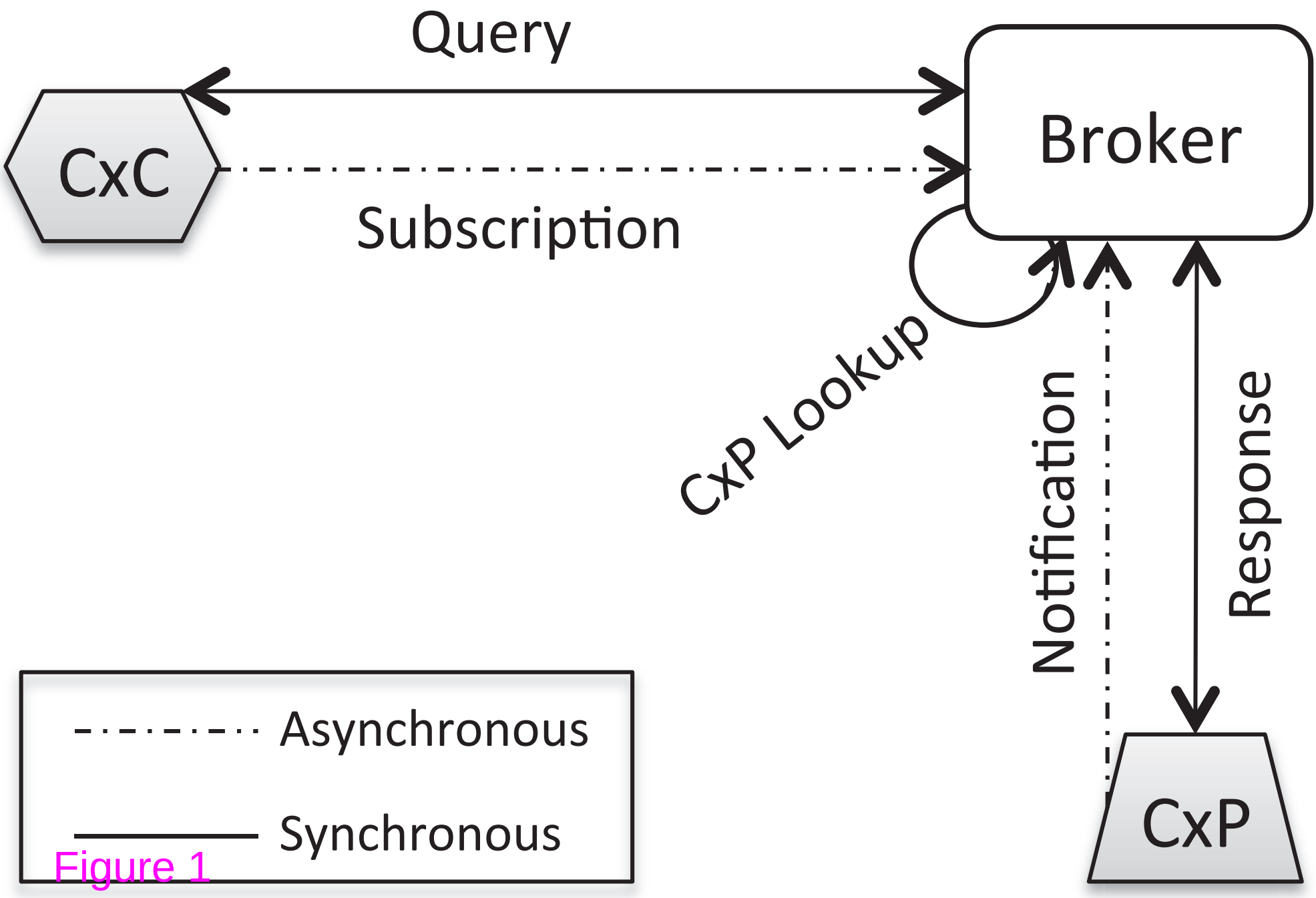
Query
CxC
Broker
Subscription
CxP Lookup
Notification
Response
CxP
Asynchronous
Synchronous

Figure 1

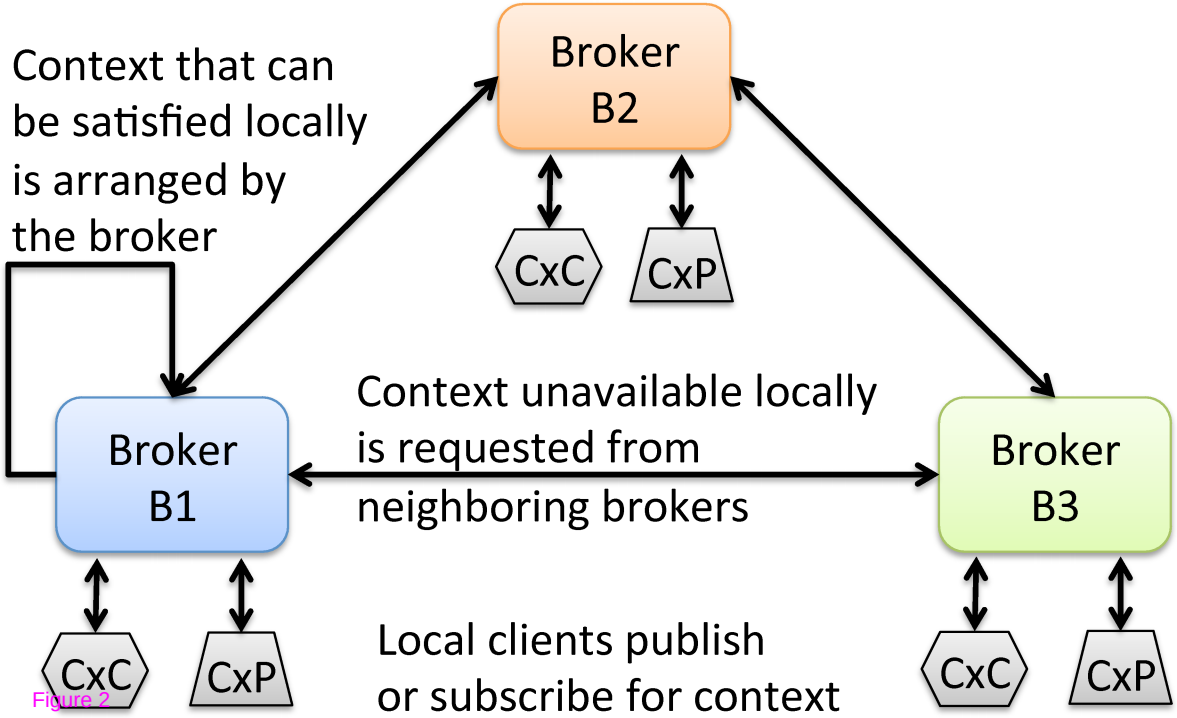
Context that can be satisfied locally is arranged by the broker
Broker B2
CxC
CxP
Context unavailable locally is requested from neighboring brokers
Broker B1
Broker B3
CxC
CxP
CxC
CxP
Local clients publish or subscribe for context

Figure 2

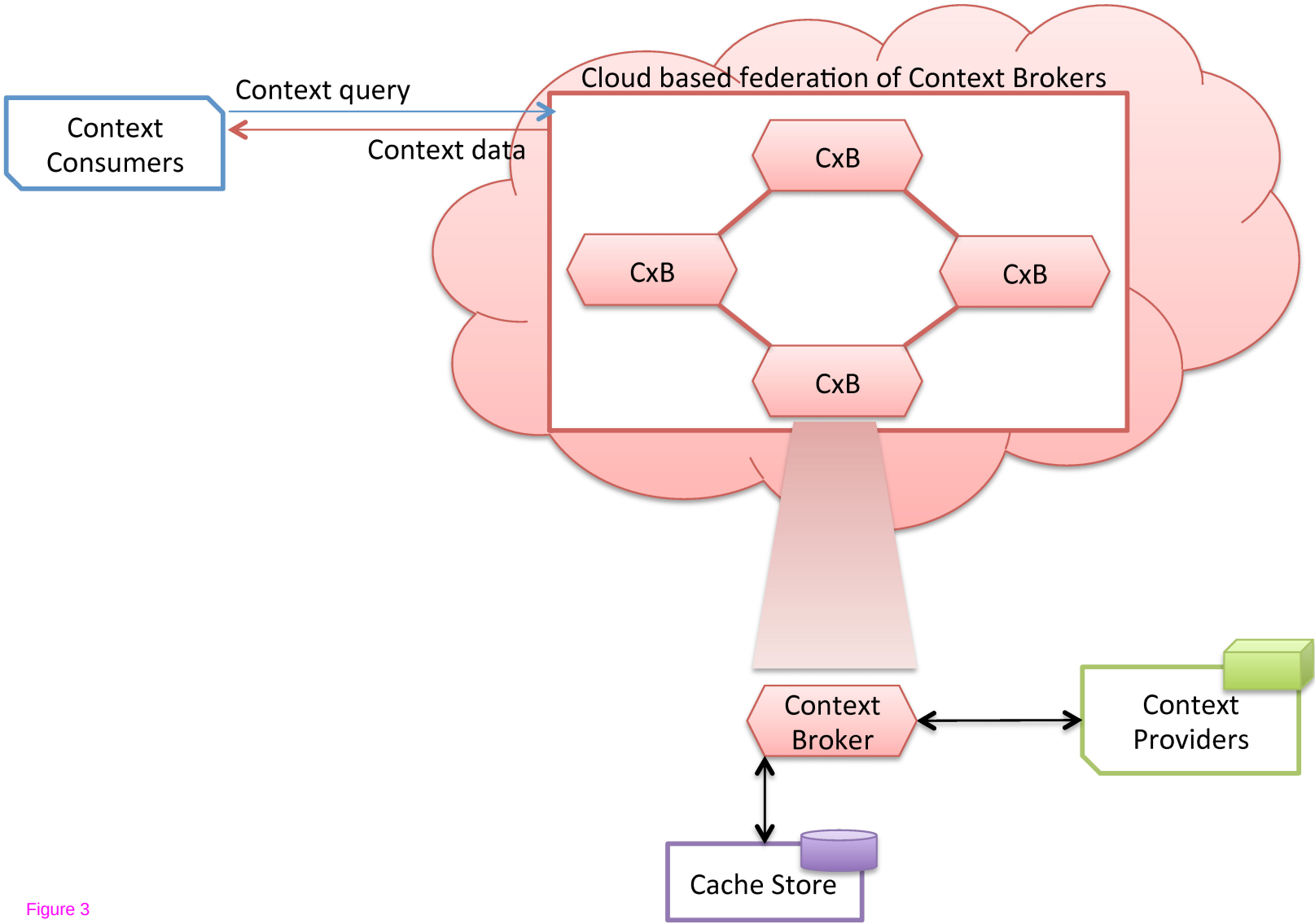
Context Consumers
Context query
Context data
Cloud based federation of Context Brokers
CxB
CxB
CxB
CxB
Context Broker
Context Providers
Cache Store

Figure 3

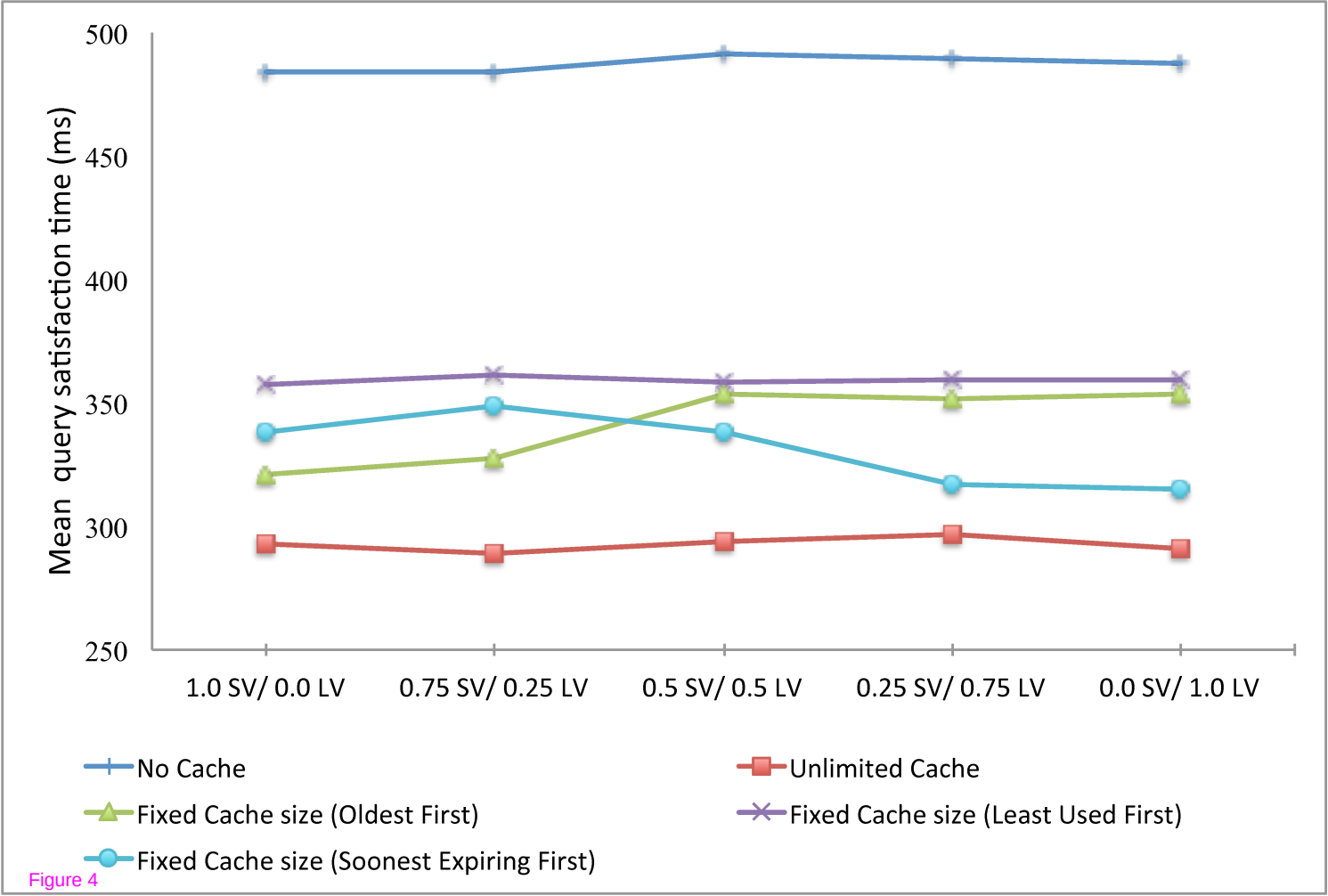
Mean query satisfaction time (ms)
500
450
400
350
300
250
1.0 SV/ 0.0 LV
0.75 SV/ 0.25 LV
0.5 SV/ 0.5 LV
0.25 SV/ 0.75 LV
0.0 SV/ 1.0 LV
No Cache
Unlimited Cache
Fixed Cache size (Oldest First)
Fixed Cache size (Least Used First)
Fixed Cache size (Soonest Expiring First)

Figure 4

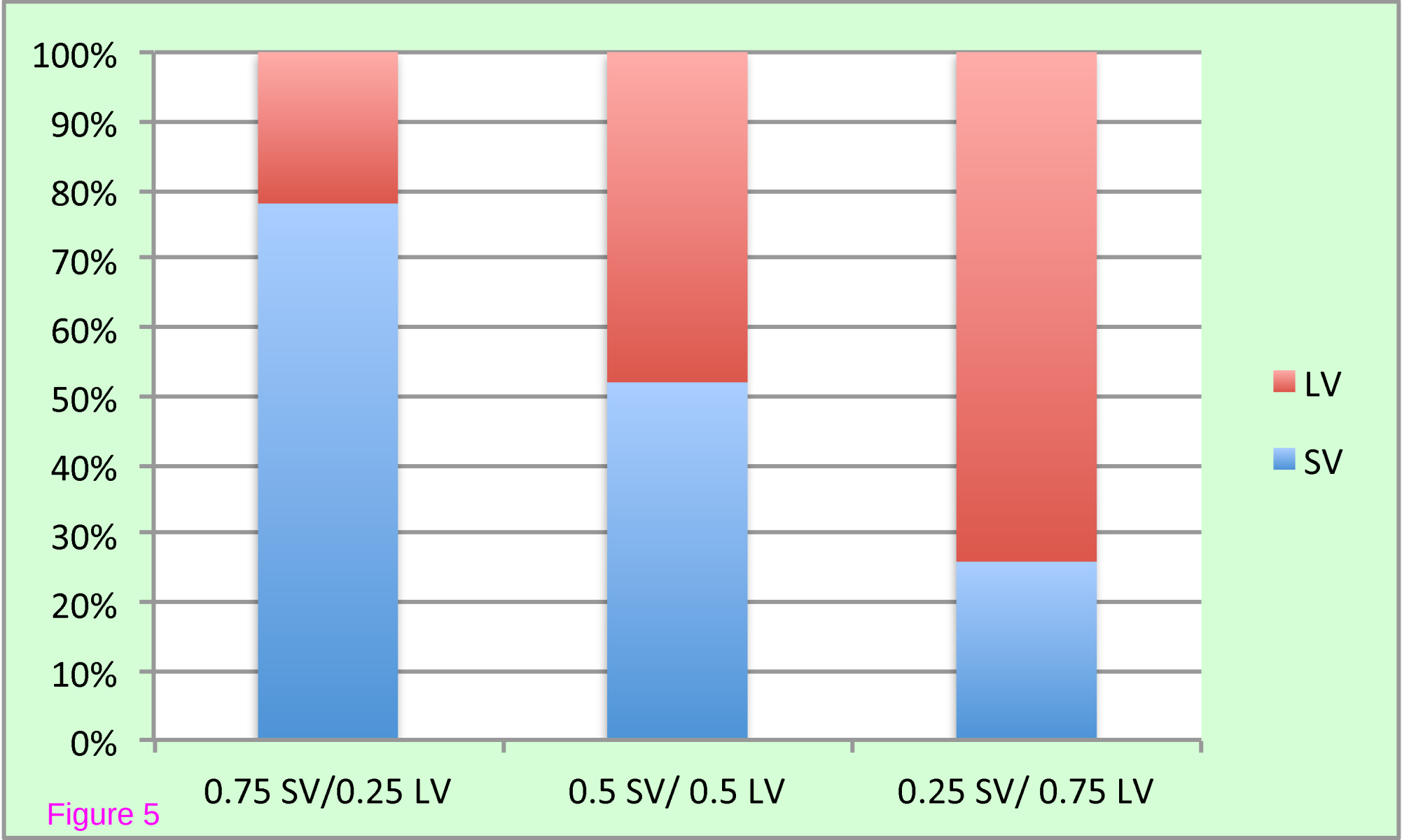
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
0.75 SV/0.25 LV
0.5 SV/ 0.5 LV
0.25 SV/ 0.75 LV
LV
SV

Figure 5

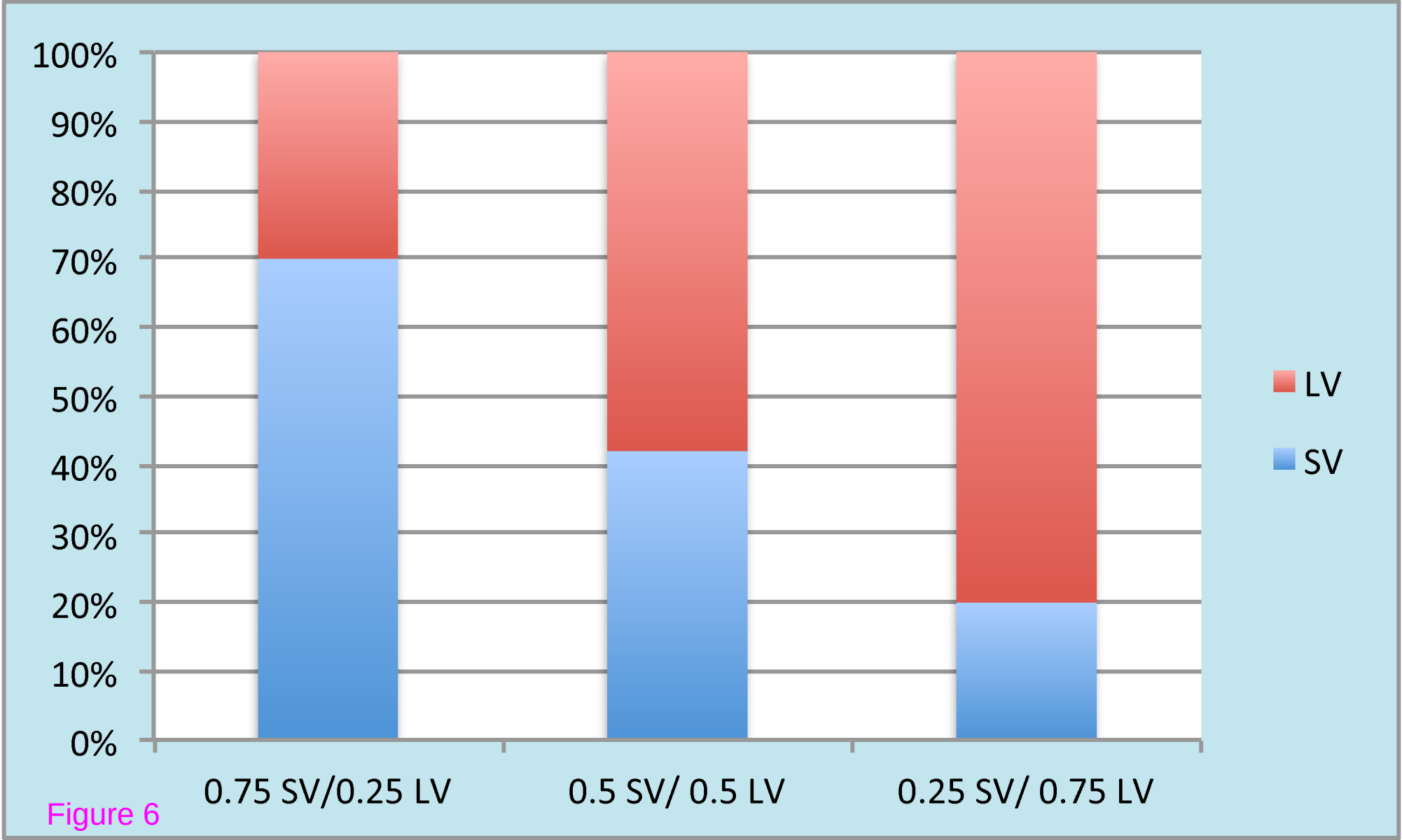
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
0.75 SV/0.25 LV
0.5 SV/ 0.5 LV
0.25 SV/ 0.75 LV
LV
SV

Figure 6

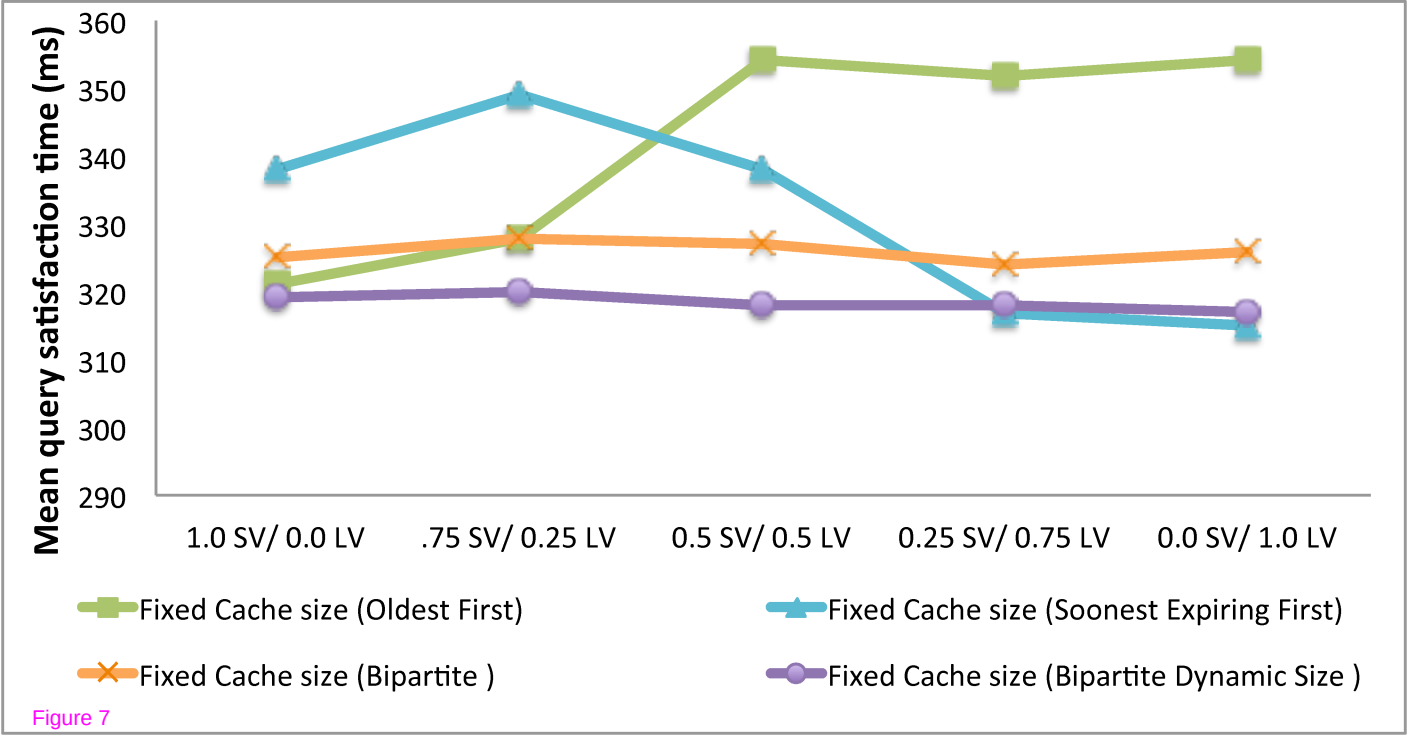
Mean query satisfaction time (ms)
360
350
340
330
320
310
300
290
1.0 SV/ 0.0 LV
.75 SV/ 0.25 LV
0.5 SV/ 0.5 LV
0.25 SV/ 0.75 LV
0.0 SV/ 1.0 LV
Fixed Cache size (Oldest First)
Fixed Cache size (Soonest Expiring First)
Fixed Cache size (Bipartite)
Fixed Cache size (Bipartite Dynamic Size)

Figure 7

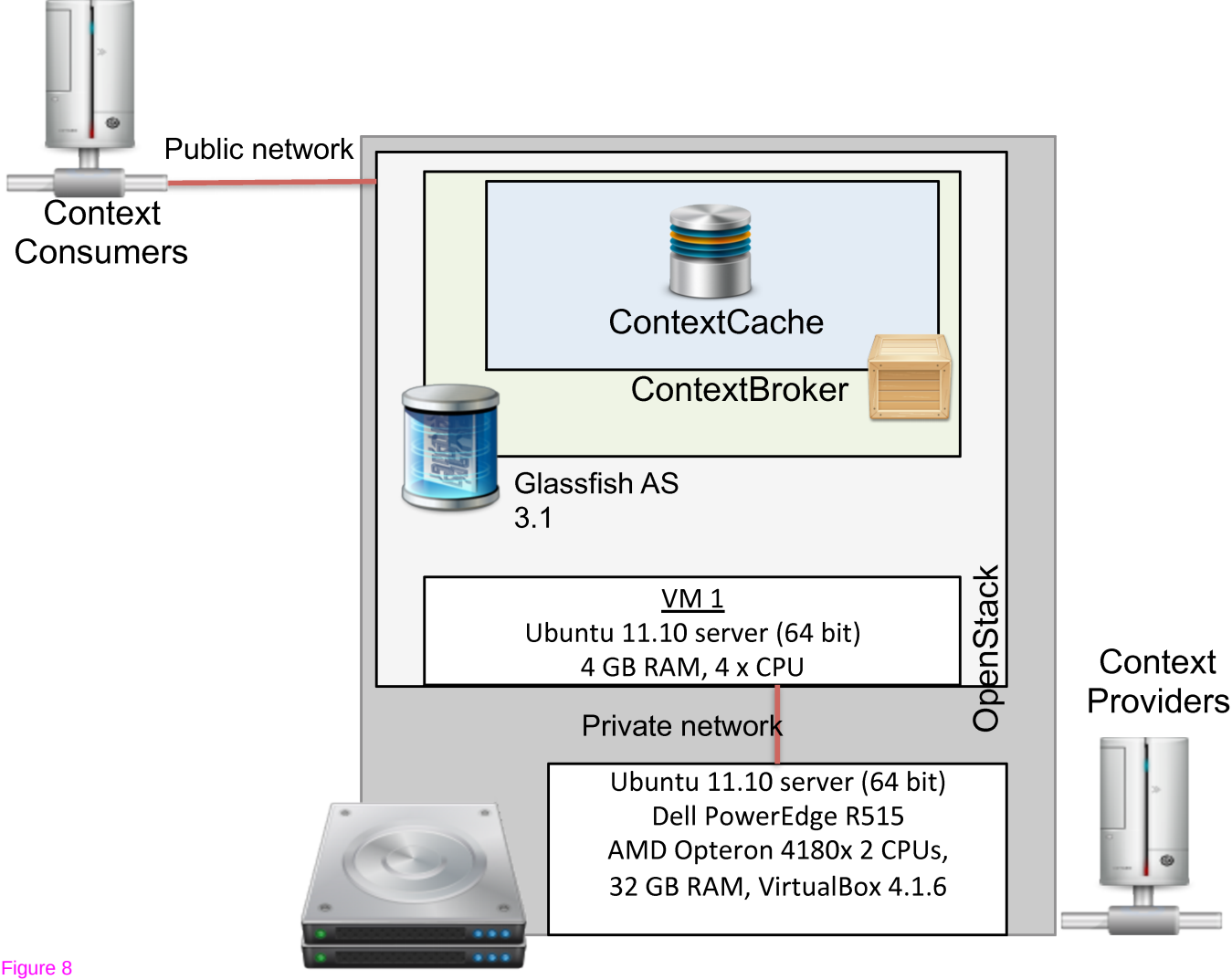
Public network
Context Consumers
ContextCache
ContextBroker
Glassfish AS 3.1
VM 1
Ubuntu 11.10 server (64 bit)
4 GB RAM, 4 x CPU
OpenStack
Private network
Ubuntu 11.10 server (64 bit)
Dell PowerEdge R515
AMD Opteron 4180x 2 CPUs,
32 GB RAM, VirtualBox 4.1.6
Context Providers

Figure 8

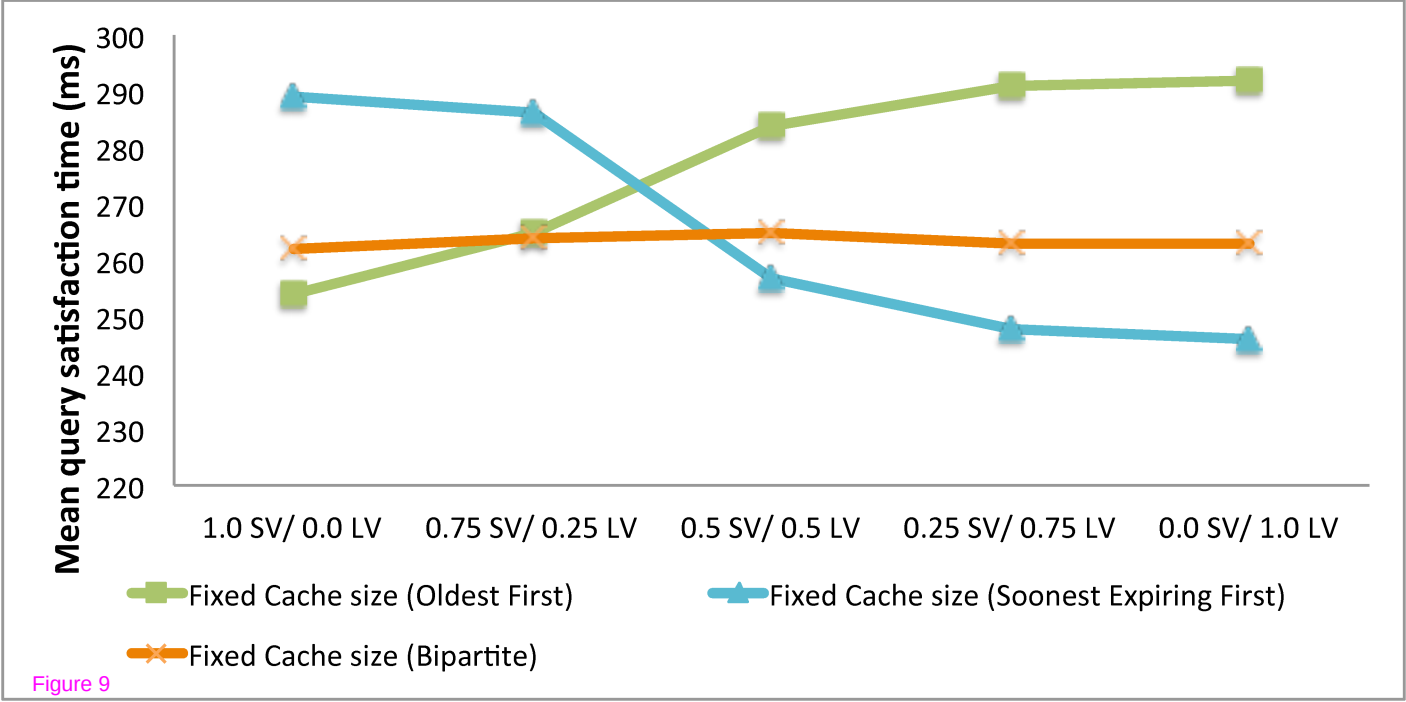
Mean query satisfaction time (ms)
300
290
280
270
260
250
240
230
220
1.0 SV/ 0.0 LV
0.75 SV/ 0.25 LV
0.5 SV/ 0.5 LV
0.25 SV/ 0.75 LV
0.0 SV/ 1.0 LV
Fixed Cache size (Oldest First)
Fixed Cache size (Soonest Expiring First)
Fixed Cache size (Bipartite)

Figure 9

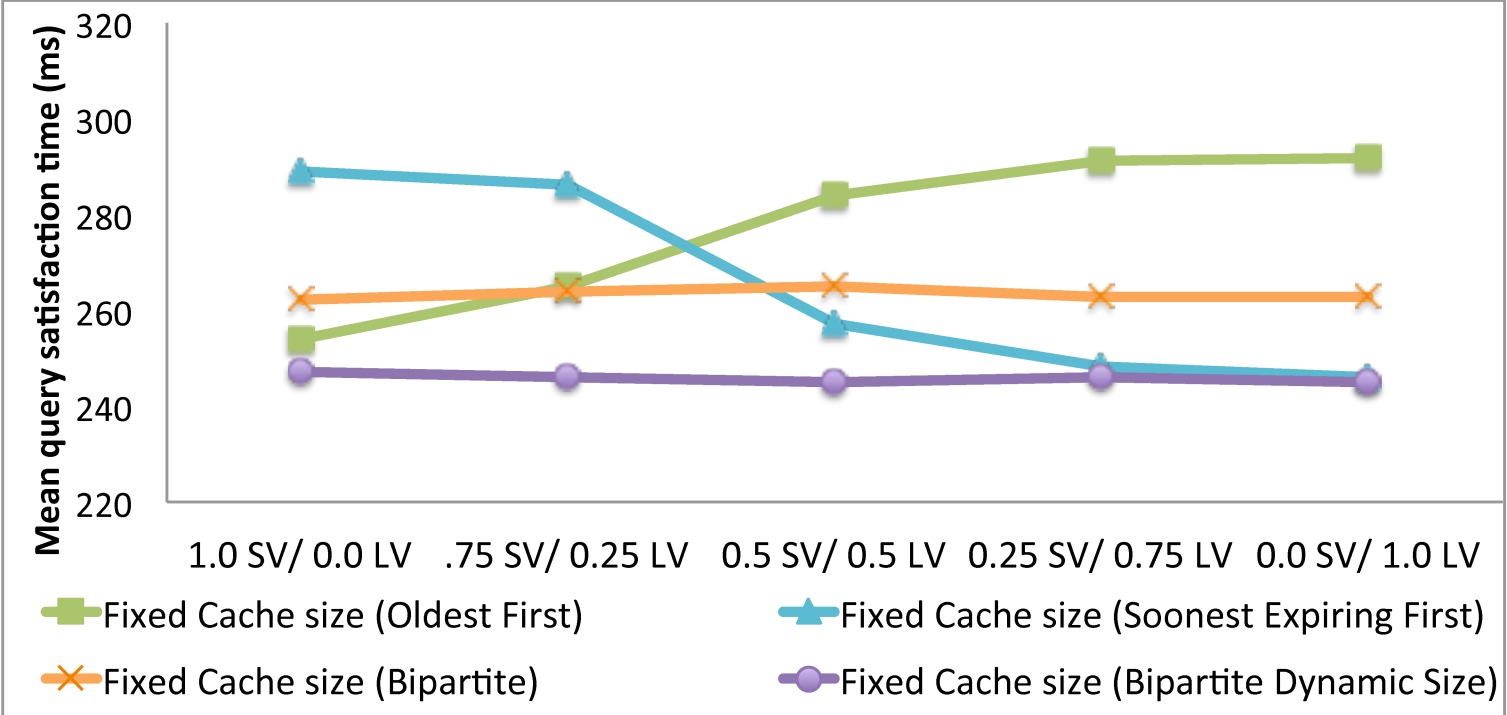
Mean query satisfaction time (ms)
320
300
280
260
240
220
1.0 SV/ 0.0 LV
.75 SV/ 0.25 LV
0.5 SV/ 0.5 LV
0.25 SV/ 0.75 LV
0.0 SV/ 1.0 LV
Fixed Cache size (Oldest First)
Fixed Cache size (Soonest Expiring First)
Fixed Cache size (Bipartite)
Fixed Cache size (Bipartite Dynamic Size)

Figure 10