The Viability of Scaling in a Distributed Spatial Cache Overlay

Alexander Gessler, Simon Hanna, Ashley Smith

Abstract—Location-based services for mobile devices, such as route-planners, access data with spatial and temporal characteristics. How to balance the workloads for location-based services is an open problem. To investigate viable load-balancing mechanisms, we designed, implemented, and measured a distributed, spatial cache overlay for two-dimensional, geographic data. In this paper, we describe an experimental setup to benchmark such a system and we discuss the challenges we faced. We measure relative scalability for different node counts, query rates and workload distributions and show that the system is able to achieve almost linear relative scalability for uniform distributions, but that load-balancing is required to ensure scalability under non-uniform or changing load. We present an outlook on future extensions to the system.

Index Terms—Location-Based Services, Distributed Systems, Caching, Spatial Cache, DHT, Load-Balancing, Scale-in.

1 INTRODUCTION

A key problem in distributed systems is the efficient allocation of workloads among nodes in the system, so as to guarantee low latency in computing or processing workloads, and to optimally take advantage of the available resources. Changing unevenly distributed workloads therefore poses a challenge.

Location-based Services are applications that utilize the geographic position of a mobile device that the user is carrying, in order to provide information and services. A classical example is a route-planning or map application for smartphones. The user sends an address as a query, and in response to the query, the application delivers the corresponding portion of the map. Inherent to such services is that they expect shifting, irregular workload patterns with respect to time and spatial position of the data requested. In our work we use a distributed caching system for spatial data that is set to adapt to such workload patterns. The system uses generic cache nodes, each of which holds a small cache for map tiles within spatial proximity to a point that we call the “cache focus” of that node. This kind of data-oriented network overlay, whose nodes correspond to caches that are partitioned spatially, is also known as a spatial cache overlay. Given these concepts of spatial proximity and partitioning, an anticipated workload can be divided in advance among multiple cache nodes.

Nevertheless, balancing the workloads remains problematic, even with a spatial cache overlay, due to the temporal and spatial variability of queries to an LBS. The amount of queries will vary, depending on the time of day or the geographic density of users. For example, fewer queries will be sent late at night or in rural areas. Moreover, this variability can be so skewed that it is difficult to model, given that extreme spikes in queries are difficult to predict. Because of the nature of variability in LBS query loads, the problem of balancing workloads in a distributed cache overlay requires an adaptive solution that scales correspondingly to non-uniform and unpredictable workloads.

To help realize such an adaptive solution, we continue the previous work of Lübbecke et al. [1], [2], who developed a novel elastic load-balancing mechanism between cache nodes organized in a distributed spatial cache overlay. Our contributions to their work consist of implementing a distributed cache overlay to verify the results originally achieved with a simulation. Another goal is to measure the effectiveness of load-balancing by dynamically adding or removing nodes (scale-in and scale-out).

The remainder of this paper is organized as follows: we establish the motivation and basis of evaluation against related approaches. Then the relevant key concepts of elastic load-balancing, scale-in and scale-out and Delaunay triangulation are elaborated in Sections 2. Next, we describe the software architecture under its system specifications and assumptions in Section 3 and its actual implementation in Section 4. The performance of the distributed spatial cache overlay, both with/without load-balancing through scale-in and scale-out and under uniform and non-uniform query distributions, is presented as results and evaluated in Section 5. We discuss the promising continuations of our work in Section 6 and conclude in Section 7.

2 FOUNDATIONS

Based on the work of Lübbecke et al. [1], [2], we partition the region into a grid overlay, consisting of nodes that cache data. The “location” of a cache node in this two-dimensional space is a cache focus. When a cache node serves as request for a part of the map that causes a local cache miss, it evicts the existing cache entry with the largest distance to the cache focus to make room for the new entry. Our grid forms a Delaunay triangulation in a 2D metric space. “Greedy routing” can be employed to route messages to the node that is closest to a certain 2D coordinate. A node that receives a message checks if any of their neighbors is closer to the location that the message is to be routed to (with respect to some metric). If so, it forwards the message to the neighbor node that is closest. Otherwise, it handles the message. We use the Delaunay triangulation specifically, as Bose and Morin [7] have proven that greedy routing in a Delaunay triangulation is guaranteed to find local optimum. The topological property that is exploited is, in fact, the triangle equation of metric space. For our grid, we use of course the Euclidean. Elastic load-balancing was one approach to balancing the load. It refers to dynamically move cache nodes around in the grid to tailor cache focus at the current load situation, while almost always preserving the Delaunay property of the underlying triangulation. It is elastic because if the grid should move back to the original layout after a short spike that causes higher load in certain areas of the grid.

Scale-in and scale-out are two methods of load balancing by adding or removing nodes, respectively. The idea is that the grid is able to automatically decide when to add or remove nodes, based on self-measured, estimated load in a particular region. This differs from the idea of adding and removing as proposed by [quote carlos]. There, the addition/removal is in conjunction with a different load-balancing method called elastic load-balancing. In their work, nodes are only added or removed at the borders of the grid, in order to preserve grid stability and the Delaunay triangulation. They rely then on their elastic load-balancing to shift cache focus to the needed location.

3 SYSTEM ARCHITECTURE

When designing our system architecture, we were given relative freedom, due to the open-ended nature of the problem. We had to meet the following system specifications and functionality:

1. A grid of cache nodes that are organized spatially.

2. Next, we describe the software architecture under its system specifications and assumptions in Section 3 and its actual implementation in Section 4. The performance of the distributed spatial cache overlay, both with/without load-balancing through scale-in and scale-out and under uniform and non-uniform query distributions, is presented as results and evaluated in Section 5. We discuss the promising continuations of our work in Section 6 and conclude in Section 7.

2 FOUNDATIONS

Based on the work of Lübbecke et al. [1], [2], we partition the region into a grid overlay, consisting of nodes that cache data. The “location” of a cache node in this two-dimensional space is a cache focus. When a cache node serves as request for a part of the map that causes a local cache miss, it evicts the existing cache entry with the largest distance to the cache focus to make room for the new entry. Our grid forms a Delaunay triangulation in a 2D metric space. “Greedy routing” can be employed to route messages to the node that is closest to a certain 2D coordinate. A node that receives a message checks if any of their neighbors is closer to the location that the message is to be routed to (with respect to some metric). If so, it forwards the message to the neighbor node that is closest. Otherwise, it handles the message. We use the Delaunay triangulation specifically, as Bose and Morin [7] have proven that greedy routing in a Delaunay triangulation is guaranteed to find local optimum. The topological property that is exploited is, in fact, the triangle equation of metric space. For our grid, we use of course the Euclidean. Elastic load-balancing was one approach to balancing the load. It refers to dynamically move cache nodes around in the grid to tailor cache focus at the current load situation, while almost always preserving the Delaunay property of the underlying triangulation. It is elastic because if the grid should move back to the original layout after a short spike that causes higher load in certain areas of the grid.

Scale-in and scale-out are two methods of load balancing by adding or removing nodes, respectively. The idea is that the grid is able to automatically decide when to add or remove nodes, based on self-measured, estimated load in a particular region. This differs from the idea of adding and removing as proposed by [quote carlos]. There, the addition/removal is in conjunction with a different load-balancing method called elastic load-balancing. In their work, nodes are only added or removed at the borders of the grid, in order to preserve grid stability and the Delaunay triangulation. They rely then on their elastic load-balancing to shift cache focus to the needed location.

3 SYSTEM ARCHITECTURE

When designing our system architecture, we were given relative freedom, due to the open-ended nature of the problem. We had to meet the following system specifications and functionality:
2. The system was flexible enough to test the grid under different load-balancing and scale-in and scale-out algorithms.

3. Our system runs remotely on a computer cluster of unspecified size and cores.

4. Our system could be started and shut down by an external GUI but is independent thereof.

5. The nodes could process and forward requests.

For our implementation language we chose Java. Our development platform consisted of JRE 1.6, Eclipse, and Git. In addition to the system specifications given explicitly, we also implemented a special administrative node, asynchronous communication between nodes, and a basic logging system to diagnose system malfunction.

3.1 Architecture

Our system can be divided into three intercommunicating interfaces, as shown in Figure 1. A GUI manager can launch or shut down the grid. Users can send requests to the grid. The grid consists of two kinds of components: one administrative (admin) node and multiple cache nodes. Node know about their neighbors and are connected by two-way communication channels. Every cache node is connected to the admin node.

3.2 Phases

Our system executes in three stages. During the initialization phase, the system is setup. The GUI deploys the admin and cache nodes on the network. Next, all cache nodes connect to the admin. When all cache nodes are connected to the admin, the admin initiates the Delaunay triangulation. The number of points in the Delaunay triangulation corresponds to the number of cache nodes. After the triangulation, the admin node notifies all cache nodes about their individual position and who their neighbors are. The cache nodes then establish connections to their neighbors. Once a node has established all its connections to its neighbors, that node is ready to process queries. When all nodes have connected to their neighbors, the initialization is finished.

After the initialization stage, the system can begin processing queries. During this stage, any number of external users can send queries, and queries can enter the grid at any cache node and thus be processed or forwarded. We assume that a query’s range is sufficiently small so as to fit into one coordinate, ensuring that a single cache node can process a single query. In contrast, Lübbe and Mitschang allow for a node and its neighbors to process partial results for a single query and then merge the results. When a cache node receives a query, it checks whether the requested data is located in its region and whether it can handle the additional workload. If yes, then the node processes it and sends data to user. If not, the cache node operates under greedy routing and forwards the query to its nearest neighbor. A query is never passed infinitely around the grid but instead, eventually reaches a cache node that can process it. This optimality of greedy routing is ensured because the nodes remain connected in a Delaunay triangulation.

4 IMPLEMENTATION

We implemented the GUI manager, spatial cache overlay, a simple logger, a message-passing-interface API, and the interface by which external computers can send queries to the overlay. A cache node keeps a local measure of capacity, at 20 queries per second. Every time it receives a query, the node checks whether it can service the query and whether it has available capacity. A logger forwards log messages from cache nodes to the GUI manager to help us diagnose remote failures. To keep this message-passing from becoming a bottleneck, we use UDP and rate-limiting.

4.1 Message Passing Interface

For communication between nodes, we adopted the Message-Passing-Interface (MPI) paradigm. We designed our own simplified MPI implementation based on Java’s standard object serialization with TCP as the underlying communication mechanism. In our MPI implementation, messages are represented by instances of classes derived from a standard base that we provide. Two nodes construct exactly one MPI connection with each other. Messages carry an internal conversation ID, thus allowing us to match incoming replies to the original messages.

4.2 Phases

4.2.1 Initialization

To launch our system, the GUI manager is started and is given the number of cache nodes and port number, as Figure 2 shows. The cache grid is then constructed in multiple stages as described below. For the sake of avoiding redundancy, we omit how every sent message will be confirmed with a message and that construction does not begin without confirmations.

1. The admin node and the cache nodes are deployed on the network as JARs and launched remotely via SSH.

2. Cache nodes initiate communication with the admin node by sending it a JOIN message. The admin node keeps track of incoming JOIN requests until a threshold number is reached. This threshold number is the initial grid size, which is also the total number of cache nodes initially deployed.

3. The admin node then performs a Delaunay triangulation, as seen in Figure 3, using the number of cache nodes as the
number of points to be triangulated. The nodes are distributed randomly.

4. Next, the admin sends ADD_TO_GRID messages to all cache nodes to inform them about their location in the grid and about their neighbors. The admin node also assigns a unique integer ID to every node. Information about a neighbor is represented by a tuple consisting of the neighbor’s position and network address.

5. After all the cache nodes have confirmed receipt of an ADD_TO_GRID message, the admin node finally sends out ACTIVATE messages.

6. Upon receiving this message, a cache node engages in a handshake with each of its neighbors. In this handshake, a node and its neighbor first compare their unique integer IDs. The node with the smaller of the two IDs acts as the server, and the other then becomes the client. The node with the client role sends out a PUBLISH_ID to each of its neighbors that have the server role. Then the nodes establish connections.

The handshake helps to address the requirements of the client-server architecture that is enforced by Berkeley sockets. More specifically, even if two cache nodes know about each other, they still cannot directly connect. Instead, they must first agree upon who acts as the client, i.e. sends a connection, and who acts as the server, i.e. accepts the incoming connection.

4.2.2 Query processing

For our purposes it was sufficient to represent a query simply as a random string tagged with a single spatial coordinate. Under the assumptions of our system architecture, the queries are small enough to be processed by a single cache node. In contrast, queries in the original system developed by Lübke and Mitschang [3] were of arbitrary complexity. They used a special query language, AWQL, to formulate queries for geographic data. We do not use map data in our study because our primary concern is the caching behavior, which is independent of the data used. We mimic the generation of queries with idle waiting. Queries can be generated by any computer outside the grid, e.g. a user device or, as in a typical web setup, a front-end web API to which mobile devices connect.

To send a query, a computer connects to an arbitrary cache node, who maintains a listening port open for this purpose. This connection is retained until a response is available. We call the cache node through which query entered the grid as the “entry node”. If that cache node cannot answer the query, the node forwards the query via greedy routing to the closest neighbor, until the target cache node is found. If the target cache node’s local load measure is exceeded, the node may move the query to one of its neighbors. With focused caching, one of the neighbors may overlap that region and be able to serve the query. Once a node has accepted the query, it generates a response. The response can be either served from cache or from accessing the data back-end in case of a cache miss. The node then sends the response back to the query’s entry node, which then gives the response to the computer who generated the query. The minimum number of network hops is two, if the query’s entry node is able to respond to the query. If the query is forwarded k times within our network, the total number of hops is 2 + 1 + k.

4.2.3 Shut-down

The admin node initiates shutdown of the system using a similar procedure as described in initialization, but with shutdown and confirmation messages.

5 RESULTS AND DISCUSSION

The basis of our evaluation was to quantify relative scalability with different kinds of workloads. As a baseline set of tests, we benchmarked performance of the distributed spatial cache overlay under uniform and non-uniform query distributions. We validated the robustness of the results by benchmarking the system on a cluster at the university. Then we measured the latency of the system with uniform and non-uniform query distributions as a comparison.

5.1 Benchmark Testing

We wrote a benchmarking program to spawn threads who generate queries at predefined intervals. For every node, k queries per second are generated. This rate corresponds to a constant time of about 1000/kms between any two successive queries. Each run of the benchmark program lasts 10 seconds, so the total number of queries sent to the grid is n * 10 * k. It is important to note that while every node in the grid receives the same number of queries, there are two cases with regard to the spatial coordinates requested by those queries:

- The uniform case. The target coordinate for each query is a uniformly randomly picked coordinate.
- The non-uniform case. The target coordinate for each query is obtained by sampling a gaussian distribution with a standard deviation of 0.18 times the map size around a fixed hotspot point. The hotspot location is arbitrarily chosen for each run, but remains unchanged during that run.

Queries were generated on a 2.8 GHz Intel Core i7 with eight logic cores, and the distributed spatial cache overlay was launched on a 2.6 GHz Intel Core i7 with four logic cores. To simulate real-world latency of physical networks on this single computer, we injected an extra delay of 2ms into message-passing between nodes. This latency, combined with the ability to run many cache node processes while threads are put to sleep, should be an effective setup for simulating a truly distributed spatial cache overlay.

5.2 Experiments

We conducted three groups of experiments, the results of which are summarized in the tables in the next section. Each of the experiment groups used a distributed grid with a constant number of nodes (n = 16, 32 and 64). For each of the groups we ran experiments with different request rates.

5.3 Results

The results of our three groups of experiments are summarized in Tables 1-3. For a given grid size n and query rate k, we give the mean and median latencies of all queries sent within the experiment time. Query rate k is defined as the number of queries per node per second. Times are given in milliseconds, and the mean and median values are computed only from the top 95 percentiles of all queries, with regard to latency.
Table 1: Grid with 16 cache nodes, no scaling-in, neighbor propagation

<table>
<thead>
<tr>
<th>k = 10</th>
<th>k = 20</th>
<th>k = 25</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uniform Distr.</td>
<td>Mean: 462</td>
<td>482</td>
</tr>
<tr>
<td></td>
<td>Median: 458</td>
<td>466</td>
</tr>
<tr>
<td>Non-uniform Distr.</td>
<td>Mean: 469</td>
<td>828</td>
</tr>
<tr>
<td></td>
<td>Median: 458</td>
<td>465</td>
</tr>
</tbody>
</table>

Table 2: Grid with 32 cache nodes, no scaling-in, neighbor propagation

<table>
<thead>
<tr>
<th>k = 10</th>
<th>k = 20</th>
<th>k = 25</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uniform Distr.</td>
<td>Mean: 482</td>
<td>573</td>
</tr>
<tr>
<td></td>
<td>Median: 471</td>
<td>505</td>
</tr>
<tr>
<td>Non-uniform Distr.</td>
<td>Mean: 475</td>
<td>689</td>
</tr>
<tr>
<td></td>
<td>Median: 463</td>
<td>515</td>
</tr>
</tbody>
</table>

5.4 Discussion

In our evaluation of our system prototype, we were looking at a distributed cache overlay as described in previous chapters. The only kind of load-balancing that we currently perform is that nodes offload queries to their neighbors if their own load as measured in queries per second exceeds a certain threshold.

In our benchmark, we were running grids with different number of nodes (given by the parameter \( n \)) and request rates (given by the parameter \( k \)).

At the time of this writing, Scaling-in has not yet been implemented into the system. We expect to add it soon, and hope to get clear improvements in the performance of the non-uniform case. In other words, the measurements described above form the base case for a further evaluation of the system.

Our hypothesis was that the system shows almost linear relative scalability with regard to the uniform case. The results we obtained suggest that the scalability is indeed almost linear, as the median values for the uniform case with \( n = 64 \) are within 20 percent of the results for \( n = 16 \), yet the node count is four times higher. Scalability seems to degrade with a higher request rate, as can be seen in the \( n = 64, k = 25 \) case. As this amounts to a total query count of 1600 queries per second, we suspect that the problem in this case is not scalability of the grid, but rather our query generator’s inability to handle this many requests and responses. This has to be confirmed in further experiments.

Another hypothesis was that the non-uniform distribution of query coordinates would have significantly higher latencies, both in terms of mean and median values. This is because in the non-uniform case, most query locations concentrate on a very small region of the map and are therefore handled by only a small percentage of the available cache nodes. The results we obtained were less clear than we were hoping for. Our measurements show a clear advantage for the uniform case if we look at mean values, but the median latency is in a similar range and sometimes even smaller. A possible explanation is that the average routing time (i.e. the number of hops) is slightly better for a non-uniform query distribution, causing good median times. High mean latencies confirm the hypothesis that the cache grid was unable to handle this many requests in time, causing some requests to be put into waiting queues.

Our evaluation was certainly limited by the lack of a real, distributed system. Both the testing cluster we were given access to and our local setup were far from having \( n \) hardware cores. Also, our ability to measure with high rates of queries is very limited as this requires massive resources on the query generator, or a distributed approach. Our measurements suggest though that a distributed cache overlay as proposed by Lübbe et al. is able to achieve almost linear relative scalability for uniform loads.

6 Future Work

6.1 Scaling-In

Our primary focus for the future will be to implement scaling-in, i.e. dynamically adding or removing nodes from the grid. An experimental implementation is underway, and we hope to repeat our experiments and to show that scaling-in can, even in the absence of any further load balancing, significantly improve performance assuming that enough resources to draw new nodes from are available.

An important concern with Scaling-in helps improve grid robustness and to keep nodes from being added or removed too frequently, given that this is an expensive operation. When designing our current system architecture, scaling-in was already a consideration, so our implementation could easily accommodate it.

6.2 Architectural Changes

6.2.1 Remove admin node

As another improvement to grid’s fault tolerance, one of the first architectural changes should be to remove the admin node. The presence of the admin node is single point of failure and is only needed for the following three purposes: (i) to re-triangulate the entire grid if the triangulation degenerates, (ii) to decide whether to add or remove a cache node, based on the aggregated workloads received from all existing nodes in the system, and (iii) to initiate shut-down. However, to each of those three functions we can propose alternate solutions that do not necessitate an admin note. For example, Lee and Lam[8] discuss protocols for distributed Delauney triangulations, which we could tailor to our system and thereby render the admin node obsolete. To initiate shut-down, we could allow an arbitrary cache node to initiate shut-down by disabling its query processing, which then propagates throughout the network via breadth-first search. A problem could arise if some nodes are processing and routing queries that cannot be further routed correctly, given that part of the cache overlay is deconstructing itself. For this reason, we suggest investigating whether such a problem could be solved by a three-stage approach where further query processing is first disabled, then a waiting stage for all pending queries, and a third one to kill all nodes.

6.2.2 Add additional local caches

To reduce query latency even further, we suggest that each cache node keep a local list of recent queries that entered the overlay via that node, along with the position of the cache node who ultimately processed the query. Then as similar queries arrive, the cache node could search its cache for that query and create an on-demand, temporary connection to the corresponding node. Assuming a constant list size, we do not expect a change in asymptotic behavior; we do, however, expect a significant decrease in query latency.

6.2.3 Organize node connections as a skip list

Organizing cache node connections as a skip list could significantly reduce the worst-case number of hops and thereby increase scalability. Consider that many hops must be taken, if the entry point of a query into the grid is topologically distant from the cache node
which handles the region containing the query’s corresponding data. The worst case scenario would occur then in a roughly square grid with \( n \) nodes and require up to \( O(\sqrt{n}) \) hops. Each of these hops has high constant cost, as it requires package decoding through the operating system, in addition to a context switch to the routing thread. While some of this overhead could be reduced by optimizing the software stack, this worst-case estimate of hops would nevertheless reduce scalability.

Thus we suggest organizing a node’s connections as a two-dimensional skip list, where cache nodes not only maintain connections to their direct neighbors, but also to some of their second-degree neighbors, i.e. the neighbors of their neighbors, and to even fewer of their third-degree neighbors, and so forth. With each node maintaining a skip list of connections, an asymptotic routing time of \( O(\log \sqrt{n}) \in O(\log n) \) is to be expected. Such extra neighbor connections could be created and dynamically maintained at relatively low cost if nodes propagated their neighbor list periodically to all of their neighbors, and if they forward any incoming neighbor lists after randomly filtering out entries. A cache node would either periodically update its knowledge of remote neighbors or update in response to the failure of a connected node. The entries in the updated list would be culled from incoming lists from neighbors. Further details and a formal proof of correctness could entail a project of its own.

7 Conclusion

In our study, we designed, implemented and evaluated a distributed spatial cache overlay that is flexible enough to allow implementations of more sophisticated load balancing schemes in future. This cache overlay serve as base for further comparisons of various load-balancing mechanisms. We showed that the system is able to scale almost linearly with respect to different grid sizes. We confirm that non-uniform workloads cause the system’s performance and scalability to degrade. We also experienced that, due to the complexity of the distributed system, making proper benchmarks to draw results was not always possible. There are many factors to consider, last but not least one’s ability to generate test queries fast enough. We have many feasible ideas for future extensions and are therefore planning to continue developing and experimenting. The implications of our work and future work may help to characterize the relative situations under which one load-balancing mechanism are more effective than another, rather than focusing on absolute effectiveness in all situations.

References


