9 Distributed Data Management II – Caching

In this section we will study the approach of using caching for the management of data in distributed systems. Caching always tries to keep data at the place where it is needed. The problem with this approach is that it may not be known in advance, which processor will be the next to access a certain memory cell. Is it nevertheless possible to find good caching strategies? When is a caching strategy "good"? Obviously, a model is needed that allows us to measure the performance of caching strategies.

9.1 The model

Let $G = (V, E)$ be an arbitrary graph, and let U be the set of all shared objects. We assume that an adversary specifies a distributed application running on the nodes of the network, i.e., the adversary initiates read and write requests to the objects at the nodes of the network G. These requests must be served by an on-line caching strategy.

We restrict the class of allowed applications specified by the adversary to be *data-race free*, i.e., a write access to an object is not allowed to overlap with other accesses to the same object, and there is some order among the accesses to the same object such that for each read and write access, there is a unique least recent write. Note that this still allows arbitrary concurrent accesses to different objects and concurrent read accesses to the same object.

A caching strategy is called *consistent* if it ensures that a read request directed to an object returns the value of the most recent write access to that object. Write accesses are assumed to be object alterations rather than overwrites. That is, a writing node cannot just build a new copy of the object from scratch, but first has to search for an existing copy of the object.

We are interested in finding caching strategies that are efficient even though they have no prior knowledge of the actions of the adversary. These strategies are allowed to migrate, create, and invalidate copies of an object. We do not allow to use encoding strategies.

Initially, each object has one copy somewhere in the network. Every message or migration of an object along an edge e increases its load (also called *congestion* in the following) by one. We want to keep the congestion as small as possible. Let σ be an arbitrary data-race free sequence of read and write requests. Furthermore, let $C_e^A(\sigma)$ be the total congestion caused at e when using the caching algorithm A for σ, let $C_A(\sigma) = \max_e C_e^A(\sigma)$, and let $C_{\text{OPT}}(\sigma) = \min_A C_A(\sigma)$. Algorithm A is called *c-competitive* if there is a constant d so that for all sequences of requests σ we have

$$C_A(\sigma) \leq c \cdot C_{\text{OPT}}(\sigma) + d \; .$$

9.2 Caching in trees

Let $T = (V, E)$ be an arbitrary tree. Our caching strategy for T works in the following way:

- v wants to read x: v sends a request to the closest node u in T that has a copy of x. Upon receipt of the request, u sends a copy of x back to v. Every node visited by the copy will store a copy of x.

- v wants to rewrite x: v sends a message with the new object for x to the closest node u in T that has a copy of x. Upon receipt of the message, u starts in invalidation broadcast to all other nodes that have a copy of x. Afterwards, it sends the updated copy to v (*without* creating a copy for itself or placing copies along the way).

The strategy ensures the following fact.

Fact 9.1 *For every object x and every time step t, the nodes that have a copy of x form a connected component.*

The fact allows us to efficiently locate the nodes that store copies of an object x: every node has a sign post that points to the last node that issued a write request. (Initially, all sign posts point to the unique copy of x.) Additionally, we set markers at the border nodes of the connected component, so that the invalidation broadcast is confined to the nodes of the connected component. Our simple caching strategy achieves the following remarkable result.

Theorem 9.2 *The caching strategy is 3-competitive.*

Proof. Since the caching algorithm treats the objects independent of each other, it suffices to show that it is 3-competitive for any single object. In order to show that the caching algorithm is 3-competitive for a single object, it furthermore suffices to show that it is 3-competitive for any sequence of read requests followed by a write request.

So consider some fixed object x and an arbitrary sequence of requests to x that consists of a sequence of read requests followed by a write request. Let v_0 be the last node that issued a write request prior to the read requests, let $v_1, \ldots, v_{k-1}$ be the nodes that isssued read requests to x, and let v_k be the node that issued the write request afterwards. Consider the smallest subtree T' in T that contains the nodes $v_0, \ldots, v_k$. Obviously, every data management strategy must use every edge in T' at least once. Our caching strategy uses each edge at most three times:

- at most once to send a read request,
- at most once to send the object back, and
- once to invalidate the copies of x due to the write request of v_k.

Thus, we obtain a competitive ratio of at most 3. $\sqcap$

9.3 Caching in meshes

Consider an arbitrary $n \times n$-mesh M. Our caching strategy for the mesh is based on a hierarchical decomposition of its nodes that is done in the following recursive way:

If M only consists of a single node, we are done. Otherwise, we partition M into two submeshes along the dimension with the most nodes (see Figure 1).

This hierarchical decomposition can be represented as a *decomposition tree* $T(M)$. Its root represents M, every internal node of $T(M)$ represents a submesh of M, every leaf a node of M, and every edge a partition of a submesh into two submeshes. Thus, $T(M)$ is a binary tree of depth $O(\log n)$. We view now $T(M)$ as a virtual network that we want to simulate by M. To be able to compare the congestion in both networks, we define the *bandwidth* $b(e)$ of an edge $e = \{v, w\}$ in $T(M)$ (where v is the father of w) as the number of edges that leave the submesh $M(w)$.

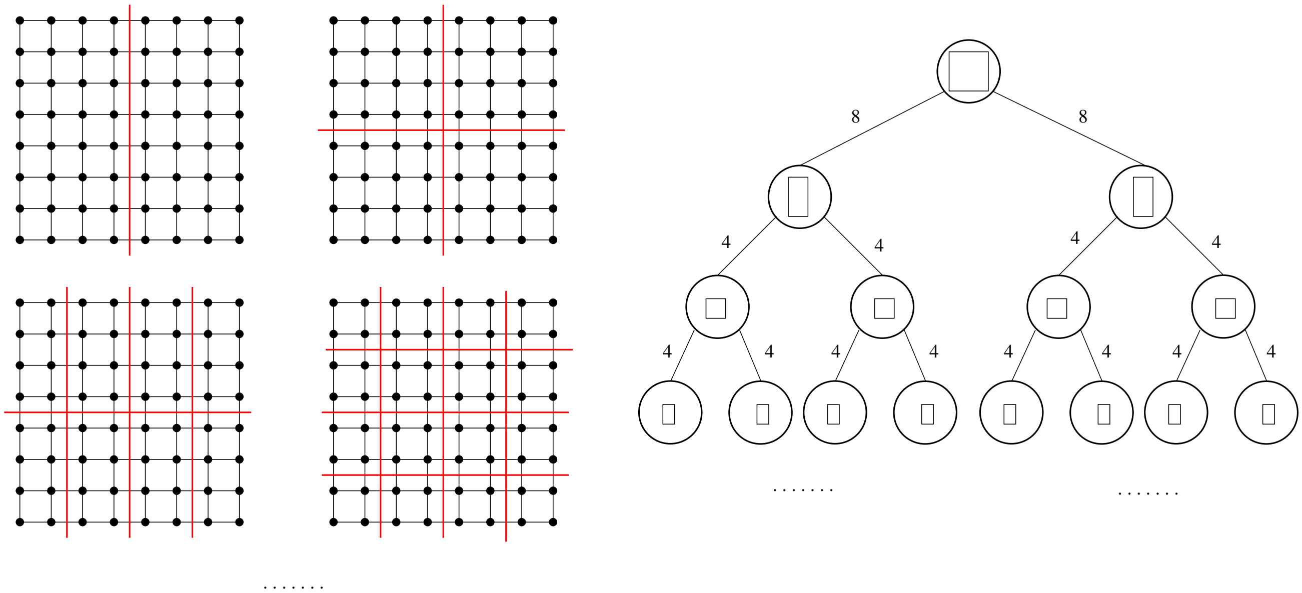

Figure 1: The decomposition of M into submeshes.

For every object x, we define an access tree $T_x(M)$ of x as a copy of the decomposition tree $T(M)$. We embed $T_x(M)$ randomly in M by mapping every node v in $T_x(M)$ to a random processor $\pi(v)$ in $M(v)$ and every edge $\{v, w\}$ in $T_x(M)$ to a path from $\pi(v)$ to $\pi(w)$ using the $x - y$ routing strategy presented in Section 3.1. This allows to simulate the movement of any message in $T_x(M)$ by moving it along the corresponding paths in M.

We apply now to $T_x(M)$ the same caching strategy that we used for the simple tree in Section 9.2. When comparing our caching strategy with an optimal *static* placement strategy (i.e. every object is stored at a single, fixed node in M), we obtain the following theorem.

Theorem 9.3 ([1]) *The caching strategy for the $n \times n$-mesh is $O(\log n)$-competitive, with high probability.*

Proof. In order to prove the above result we investigate two caching simulations. First, we consider the problem of simulating an optimal caching strategy in the mesh M by the tree $T(M)$, and then we consider the problem of simulating a caching strategy in $T(M)$ by the mesh M.

In order to simulate an optimal caching strategy in the mesh by $T(M)$, we use the simple strategy that whenever a request is sent from node v to w in M, we send it along the unique path from the leaf representing v to the leaf representing w in $T(M)$.

In order to simulate a caching strategy in $T(M)$ by M, we use the strategy already explained above: For every object x we embed $T_x(M)$ randomly into M, and whenever a request is sent along an edge $\{v, w\}$ in $T_x(M)$, we send it along the $x - y$ path from $\pi(v)$ to $\pi(w)$ in M.

Let $C_{\mathrm{OPT}}(M)$ be the best possible congestion achievable in the mesh M for the given application and let $C_{\mathrm{OPT}}(T(M))$ be the congestion caused in $T(M)$ when simulating this best possible strategy by $T(M)$. Then we get the following result.

Lemma 9.4 $C_{\mathrm{OPT}}(T(M)) \leq C_{\mathrm{OPT}}(M)$.

Proof. Let e denote an edge of $T(M)$ with bandwidth $b(e)$, and let $C(e)$ be the congestion caused by messages traversing e when using the simulation strategy above (i.e. the total number of messages traversing e is $b(e) \cdot C(e)$). Any message that crosses $e = \{v, w\}$ in $T(M)$ corresponds to a message that either has to leave or enter the submesh $M(w)$ in M. Since there are only $b(e)$ edges leaving $M(w)$, there must be an edge in $M(w)$ with congestion at least $(C(e) \cdot b(e))/b(e) = C(e)$. Thus, the maximum congestion over all edges in $T(M)$ is at most the maximum congestion over all edges in M, which proves the lemma. $\sqcap$

The next lemma gives an upper bound on the expected congestion caused by simulating the access tree strategy on M. In the following, let $C_T(e)$ denote the congestion caused at e for simulating $T(M)$.

Lemma 9.5 *For any edge e of M,* $\mathrm{E}[C_T(e)] = O(\log n \cdot C_{\mathrm{OPT}}(T(M)))$.

Proof. Let h denote the height of $T(M)$ and let $C_\ell(e)$ denote the congestion caused at e due to the simulation of edges on level ℓ of $T(M)$, $1 \leq \ell \leq h$. We show that $\mathrm{E}[C_\ell(e)] = O(C_{\mathrm{OPT}}(T(M)))$ for all $\ell \in \{1, \ldots, h\}$, which yields the lemma as $h = O(\log n)$.

Consider some fixed level ℓ. Let v be a node of $T(M)$ on level $\ell - 1$ such that $M(v)$ includes edge e. (If such a node does not exist then $\mathrm{E}[C_\ell(e)] = 0$.) Let v' be one of the two children of v. We bound the expected congestion on e due to the simulation of the tree edge $e_T = \{v, v'\}$ on level ℓ of $T(M)$.

Notice that every two levels in $T(M)$ the side length of the submeshes reduces by a factor of two. Hence, $M(v)$ is a $\Theta(n/2^{\ell/2}) \times \Theta(n/2^{\ell/2})$-mesh. Suppose that the congestion at e_T is $C(e_T)$. Since e_T has a bandwidth of $\Theta(n/2^{\ell/2})$, this means that $\Theta(C(e_T) \cdot n/2^{\ell})$ requests pass through e_T. Since every object x chooses a random place for v and v' in $M(v)$ and the $x - y$ routing strategy only goes through e if v is in the row of e or v' is in the column of e, the probability that the path for object x simulating $\{v, v'\}$ moves through e is equal to $\Theta(2^{\ell/2}/n)$. So for $\Theta(C(e_T) \cdot n/2^{\ell/2})$ requests using e_T the expected congestion at e is

$$\Theta(C(e_T) \cdot n/2^{\ell/2}) \cdot \Theta(2^{\ell/2}/n) = \Theta(C(e_T)) \ .$$

Thus, $\mathrm{E}[C_\ell(e)] = O(C(e_T)) = O(C_{\mathrm{OPT}}(T(M))$, which proves the lemma. $\sqcap$

Combining the two lemmata, we get $\mathrm{E}[C_T(e)] = O(\log n \cdot C_{\mathrm{OPT}}(M))$ for all edges e in M. One can also show that this bound holds with high probability, using the fact that for an object x with k write accesses the congestion in OPT caused for object x must be at least $k/4$ because all of these requests must update x at the same, static location. $\sqcap$

One can also extend the theorem to the case that OPT is allowed to use dynamic data management (i.e. the location of object x can change) by choosing a new random location of a node v in $T_x(M)$ each time a write request passes v (and updating the pointers of its neighbors in $T_x(M)$ correspondingly).

In addition to the upper bound, one can show the following result (see also Section 3).

Theorem 9.6 *Any online caching strategy on the $n \times n$-mesh has a competitive ratio of* $\Omega(\log n)$.

Hence, the online caching strategy presented above is asymptotically optimal.

9.4 Caching in arbitrary networks

One can observe from the proof for the mesh that there is a general way of coming up with good online caching strategy: Given some network G, try to come up with a decomposition tree $T(G)$ for G so that

- $C_{\mathrm{OPT}}(T(G)) \leq C_{\mathrm{OPT}}(G)$, i.e. an optimal strategy in G can be simulated by $T(G)$ with at most the congestion the optimal strategy needs, and
- for any edge e of G, $\mathrm{E}[C_T(e)] = O(\gamma \cdot C_{\mathrm{OPT}}(T(G)))$, i.e. there is a way to simulate a caching strategy in $T(G)$ by G so that the congestion increases by a factor of at most γ.

The first item is easy to achieve, because no matter how G is decomposed to form $T(G)$, Lemma 9.4 is always true. The difficult part is the second item. Here, it is important to come up with a tree $T(G)$ so that

- the height h of $T(G)$ is low, and
- for every edge e in G and every level ℓ in $T(G)$ it holds that $\mathrm{E}[C_\ell(e)] = O(\beta \cdot C(e_T))$ for some low β, where e_T is an edge from level $\ell - 1$ to ℓ in $T(G)$ whose $(\ell - 1)$-level node contains e.

In this case we would get $\gamma = O(h \cdot \beta)$. For further information in this subject see [3, 4].

One may ask whether it is always possible to find decompositions so that h and β are polylogarithmic. Surprisingly, Räcke recently showed that this is possible:

Theorem 9.7 ([2]) *For every network G with non-negative capacities there is an online caching strategy that achieves for every application a congestion of $O(C_{\mathrm{OPT}}(G) \cdot polylog(n))$.*

Hence, even if no knowledge is available about the requests issued by the processors, efficient online caching is possible for arbitrary networks.

References

[1] B. Maggs, F. M. auf der Heide, B. Vöcking, and M. Westermann. Exploiting locality for networks of limited bandwidth. In *Proc. of the 38th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 284–293, 1997.

[2] H. Räcke. Minimizing congestion in general networks. In *Proc. of the 43rd IEEE Symp. on Foundations of Computer Science (FOCS)*, 2002.

[3] B. Vöcking. *Static and Dynamic Data Management in Networks*. PhD thesis, Dept. of Mathematics and Computer Science, University of Paderborn, October 1998.

[4] M. Westermann. *Caching in Networks: Non-Uniform Algorithms and Memory Capacity Constraints*. PhD thesis, Dept. of Mathematics and Computer Science, University of Paderborn, November 2000.