11 Distributed Data Management II

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.

11.1 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 $C_{\mathrm{OPT}}(\sigma) = \min_A C_A(\sigma)$. Algorithm A is called *c-competitive* if there is a constant d so that for all sequences of requests σ we have

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

11.2 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 11.1 *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 11.2 *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. □

11.3 Caching in meshes

Consider an arbitrary d-dimensional 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 $O(\log n)$, where n is the number of nodes in M. We now view $T(M)$ 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* of an edge e in $T(M)$ as the number of edges that leave the submesh $M(e)$.

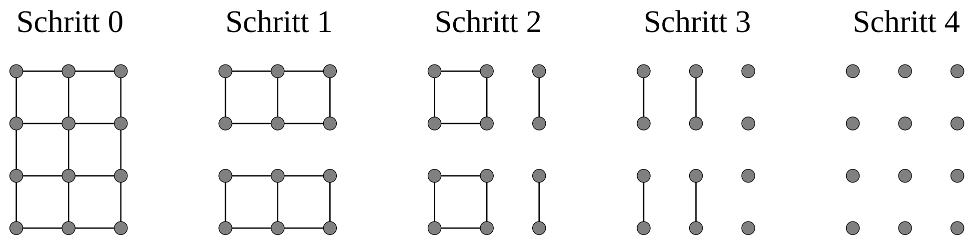

Figure 1: The decomposition of M into submeshes.

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, i.e., every node v in $T_x(M)$ is mapped to a random processor in $M(v)$. All messages that are sent along an edge in $T_x(M)$ are sent along the unique shortest path in M that uses first dimension 1, then dimension 2, and so on. We apply now to $T_x(M)$ the same caching strategy that we used for the simple tree in Section 11.2. Then we obtain the following theorem that will not be shown here.

Theorem 11.3 *Our caching strategy for the d-dimensional mesh with n nodes is $O(d \log n)$-competitive, w.h.p.*

The high competitive ratio is due to the fact that the paths used for the simulation of the edges in the $T_x(M)$ trees cause a high congestion. On the other hand, one can show that for constant d the competitive ratio is the best that can be achieved by *any* on-line data management strategy.

11.4 References

- B. Maggs, F. Meyer 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)*, pp. 284–293, 1997.
- B. Vöcking. *Static and Dynamic Data Management in Networks.* Ph.D. thesis, Dept. of Computer Science, Paderborn University, October 1998.