Data-aware connectivity in mobile replicated systems

João Barreto João Garcia Luís Veiga Paulo Ferreira
INESC-ID / Technical University of Lisbon
joao.barreto@inesc-id.pt jog@gsd.inesc-id.pt luis.veiga@inesc-id.pt
paulo.ferreira@inesc-id.pt

ABSTRACT

Optimistic replication is a fundamental technique for supporting concurrent work practices in mobile environments. However, due to sudden and frequent transitions to weakly connected situations, user experience when accessing replicated data is poor and discourages users from using the replication service. While most research on optimistic replication assumes weak connectivity as a fixed imposition of the environment, weak connectivity often results from a user option of disconnecting available connections in order to reduce battery and/or monetary cost.

This paper argues that such a choice can be considerably optimized if driven by the system, rather than the user. For that, we propose to rely on the accurate knowledge the system can have about the replicas it stores, along with the information about the available connections and the corresponding costs. We introduce the notion of *data-aware connectivity*, where the system regulates which available connections to enable, with the intent of (i) ensuring acceptable quality of accessed replicated data, (ii) at minimal connectivity cost. We propose a system for data-aware connectivity, which integrates well with existing operating systems and replicated data infra-structures.

Keywords

D.4.2 Storage Management, D.4.4 Communications Management

1. INTRODUCTION

Optimistic replication [19] is a fundamental technique for supporting concurrent work practices in mobile environments. As collaboration through these environments becomes popular (e.g. by using asynchronous groupware applications, or distributed file or database systems, and, in the future, collaborative wikis), the importance of effectively supporting access to local consistent replicated data grows.

Optimistic replication enables replica access without *a priori* synchronization with the remaining distributed sites, therefore operating *anytime and anywhere*. However, such a high availability does not necessarily guarantee acceptable *quality* of the data that users access. Whereas, in cases of strong connectivity, optimistic replication can seamlessly behave like pessimistic replication, providing access to fresh data and committing local updates almost immediately to the remaining remote replicas, the same does not hold in situations of poor or no connectivity. In the latter case, the current value of some replicas may be out-of-date or conflicting with the values that other distributed replicas hold.

Unfortunately, when one considers a personal mobile device that travels together along with its owner, the transition from one connectivity extreme to the other can be very sudden and frequent. For instance, when the user leaves his office building to travel to another building where she will have a work meeting; or when the user disconnects the UMTS antenna during a train trip in order to save battery.

Not surprisingly, the overall user experience when accessing replicated data can be very uncomfortable and discouraging. Users accessing replicated data in weakly connected situations will often face disruptive cases such as later learning that their work was aborted because it was performed upon old values (and a more recent, conflicting version had already committed elsewhere). In reaction, most users will either: (i) not access the optimistic replicas when they sense that connectivity is not sufficiently high; or (ii) will spend considerable battery and/or money (e.g. through expensive public WI-FI hotspots or UMTS connections) to enforce strong connectivity in longer periods. Nevertheless, none is a good option. The former discards the inherent high availability of optimistic replication. While the latter tends to have a high battery/monetary cost for connectivity that is often stronger that effectively needed.

To the best of our knowledge, all research on optimistic replication for weakly connected environments assumes such situations of weak (or no) connectivity are fixed impositions of the environment; hence, their goal is to ensure the highest quality of service, despite such an assumption [19].

However, connectivity in today's mobile networks is not as immutable as commonly modeled. Most mobile devices are capable of networking through a number of network technologies (such as IrDA, Bluetooth, WiFi or UMTS) to different available networks that may be available at each instant, with distinct ranges and bandwidths, but also different costs in terms of battery consumption or access price. Therefore, most situations of poor or no connectivity are not inevitable. Rather, they result from an explicit user choice of disabling better connectivity options for lower bat-

MobiDE'09, June 29, 2009, Providence, RI, USA.

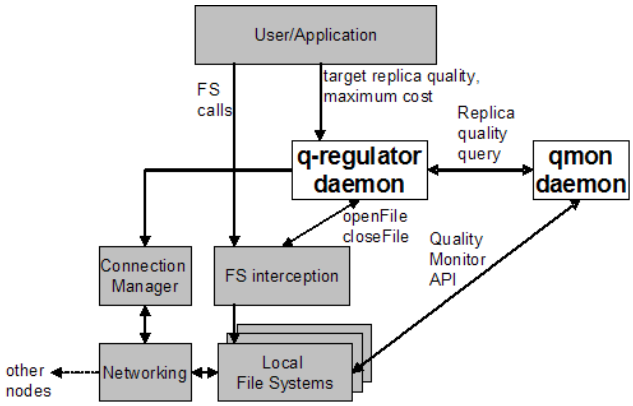

Figure 1: System architecture, based on the *qmon* and *q-regulator* daemons.

tery/monetary costs.

In this paper we argue that such a choice can be considerably optimized if taken by the system, rather than by the user, with the main intent of: (i) ensuring acceptable quality of accessed replicated data, (ii) at minimal connectivity cost. We call such a principle *data-aware connectivity*. This paper proposes a system for supporting data-aware connectivity. It relies on the accurate knowledge that local replication managers can provide about their replicas, along with information about the available connections and the corresponding costs. Furthermore, our system is generic enough to be applicable to a large class of state-of-the-art replicated systems; not only optimistic but also pessimistic ones.

Figure 1 depicts the overall architecture, which is based on two central mechanisms. A *quality monitor* continuously estimates an indicator of the current quality of each replica that is accessible in the local device. Such a measure can guide the user when deciding which task (hence, which working set to access) to pursue at a given moment, encouraging the user to carry out the task that, in the current connectivity level, accesses data with higher quality. Such a measure takes aspects such as recent update history, data freshness and accessibility to other replicas into account.

In cases where the user decides to access data with insufficient quality, a *connectivity regulator* helps her obtain enough connectivity to push data quality of a given working set to a desired level. This mechanism relieves the user from the responsibility of deciding which connections to enable, and for how long to keep them active. Instead, using knowledge about available potential connections and their cost, our mechanism tries, at each moment, to keep as few and as cheap connections as possible while achieving the desired data quality of the current working set.

2. REPLICA QUALITY

Ideally, when some user accesses a given replica, the involved data should be fresh (meaning the it reflects the most recent change to any replica of the same object), consistent with the other replicas (according to some reasonable consistency criterion, such as sequential consistency [16]), and any writes that the user issues on that replica should rapidly commit.

Of course, in weakly connected environments, we are not always able to ensure such requirements. Updates may take long to propagate to all replicas; hence, replicas that are poorly connected to replicas issuing updates will not yield the freshest value of the object. Furthermore, in some cases, yielding the most recent update does not mean that the resulting value is consistent system-wide, as other older, yet conflicting, updates may eventually commit; therefore causing the latest (freshest) update to abort. Finally, current connectivity to other replicas strongly affects a replica's ability to commit the tentative updates that its users issue. In situations where a replica has good connectivity to enough replicas (for instance, a quorum of replicas, in case of quorum-based protocols), the optimistic replication protocol will behave very similarly to a pessimistic protocol: an update will commit almost immediately, after the connected replicas exchange a small number of messages (assuming the probable case where no concurrent updates occurred in such a short period). In contrast, a replica in a remote partition will only be able to commit updates once connectivity to the main partition exists, which can take a long time to happen. Moreover, during such a long period, the probability of conflicting updates that commit concurrently with the former update (hence forcing it to abort) grows.

Therefore, when considering different replicas (of different objects) that may exist in some site, there is an implicit notion of quality of their data. It is determined by the above three dimensions: freshness, consistency and possibility of rapid commitment. Among others, a number of criteria contribute to such dimensions of quality:

1. Time since last synchronization (possibly epidemically) from every other replica:

   A lower value means that the replica reflects more recent updates, hence it is fresher.[1]

2. Number of local tentative updates:

   Since tentative updates are not guaranteed to be consistent, the more tentative updates affect the current replica value, the higher its divergence with the value that is currently consistent system-wide.[2]

3. Commit weight of currently missing from accessible replicas:

   Different commitment protocols rely on different weight distributions for agreeing on commitment of each new update (e.g. primary commit [11, 22] assigns full weight to a differentiated replica, and no weight to the remaining replicas; whereas weighted voting protocols [18, 10] distribute weight by a larger set of replicas, according to some coterie [9]). Naturally, the higher the total weight of the set of currently accessible replicas, the more probable that an issued update will commit successfully and rapidly.

4. Number of known concurrent updates:

   Issuing an update and having it committed depends strongly on the current update contention on the object. Evidently, the more tentative updates are known to be pending for commitment, the less probable it

[1]This is similar to the *staleness* criterion in TACT [23].
[2]Equivalent to TACT's *order error* [23].

is that a newly issued update that conflicts with the former will commit (and cause the former to abort).

5. Recent update activity by other replicas to the object:

   By the locality principle [6], we know that, if an object has been updated recently, it is probable that it will be written again in a close future. Hence, even when a replica is not aware of other concurrent tentative updates, the later other replicas issued tentative updates on the object, the more likely concurrent updates are to exist. Hence, for the same reasons as above, recent update activity by other replicas means a lower probability of committing new tentative work that the local replica may issue.

Each criterion above defines a dimension of the complex notion of replica quality. In specific contexts, our model can be transparently enlarged with further criteria (e.g. numerical error [23]).

The above criteria deserve two remarks. Firstly, among other factors, all the above criteria depend on the available connectivity to other replicas. If some replica currently exhibits bad indicators regarding any of the above criteria, that can be solved by increasing the connectivity of the local replica to a sufficiently larger set of replicas (and, evidently, letting the underlying replication protocol disseminate and commit updates through such connections).

As a second remark, most of the criteria are relatively intuitive to the attentive user. For example, consider a user sharing some files with other colleagues through a CVS repository. That user will often tend to work only when the CVS server is accessible (therefore maximizing criterion 3) and, from time to time, will commit her updates in order to minimize criterion 2.

However, the user estimate on replica quality can be very error-prone for a number of reasons. Firstly, it depends on events that are not always visible to the human user (e.g. events such as temporary disconnection of the CVS server, or other user committing concurrent work on the CVS server, are not immediately visible to the user). In contrast, such events are more easily trackable by the underlying replication layer.

Secondly, some criteria of replica quality are too complex to evaluate by the human user. For instance, in weighted voting replicated systems, knowing if the currently accessible network partition has sufficient weight to hold a quorum is not trivial, as intricate weight distributions may be employed [9], which can vary dynamically [10]. Thirdly, replica quality is rarely valuable when evaluated for individual replicas. Most user tasks require accessing sets of replicas, each of which with possibly different quality indicators. Determining which is the *working set quality* regarding a given user task requires evaluating each object that is expected to be accessed in the context of the task. A user estimation of such aggregate quality will frequently amplify errors, or neglect poor-quality objects whose quality affects the overall working set quality.

Hence, it should be the underlying replication layer, not the human user, to quantify replica quality, providing much more accurate (hence, valuable) estimates to the user. In the next sections we will discuss how the system can quantify replica quality and make users aware of it.

3. MONITORING REPLICA QUALITY

In order to maintain and expose replica quality in some device, each device runs a single *quality monitor*, called *qmon*. *qmon* is responsible for monitoring the quality of the local replicas. For simplicity, hereafter we focus on the case of where replicated objects are files.[3]

A replica can correspond to a regular local file (in other words, a single-replica file), or to a logical file that is replicated at other sites. Furthermore, each replica that *qmon* monitors can be maintained by one of different file systems (either local or replicated).

Using *qmon* is simple. When a local file system volume is mounted, it should notify *qmon* of the existence of that volume, identifying it with a locally unique identifier. In result, *qmon* replies with a set of pointers to callback functions that should be called by the local file system running the volume when certain events occur. Table 2 presents such functions, which we will describe next.

If this is the first time that a volume registers its presence to *qmon*, then it should tell *qmon* about the replicas currently stored in the volume by calling *registerReplica*. This function receives a globally unique file identifier (which identifies the logical file, which can be replicated at one or more replicas) and a replica identifier, which must be unique among the set of replicas of the logical file. The fact that *qmon* identifies replicas in such a location-independent manner allows the replica to freely change its location and/or name within the volume directory tree, without the need to notify *qmon*. Furthermore, a third parameter of *registerReplica* includes information about the current set of replicas of the logical file. More precisely, for each such replica, it includes its identifier along with the network address of the site that maintains the replica (e.g. its IP address) and the replica's weight in the commitment protocol. Such information will allow *qmon* to reason about connectivity among the replica set when determining the corresponding quality indicators.

qmon stores the parameters passed to *registerReplica* persistently, associated to the volume's identifier. Hence, when some previously registered volume is unmounted and then mounted again, *qmon* can simply recover the replica state associated with such a volume.

Such a state evolves as each volume creates or discards replicas (calling *registerReplica* and *unregisterReplica*, respectively). Furthermore, when the volume knows of changes to some replica set, or to changes to the weight distribution among such a set, it can update such information by calling *updateReplicaInfo*.

The volume should also notify *qmon* of other events that can affect replica quality. Namely, when some synchronization from remote replicas occurs, *hasSynchronized* tells *qmon* that the local replica is now up-to-date relatively to a (possibly partial) set of remote replicas. Furthermore, when the number of tentative updates that the current replica value reflects changes (e.g. when a new update is issued locally; when a tentative update is received from other replica and applied to the local value; or when local tentative updates either commit or abort), function *setTentativeLocalUpdates* is called to notify *qmon*. Finally, when the local

[3]As it will become clear next, this is without loss of generality, as the ideas proposed in the following sections can be transparently applied to other kinds of replicated objects.

Function	Description
`registerReplica(fileId, repId, replicaListInfo)`	Registers a new local replica of a file.
`updateReplicaInfo(fileId, replicaListInfo)`	Updates replica information list for a given file.
`unregisterReplica(fileId, repId)`	Un-registers a replica.
`hasSynchronized(fileId, repId, replicaIdList)`	Replica has become fresh with regard to a set of remote replicas.
`setTentativeLocalUpdates(fileId, repId, numberOfUpdates)`	Sets number of tentative updates that local replica value reflects.
`concurrentUpdatesKnown(fileId, repId, replicaIdList)`	Other replicas have issued concurrent updates.
`qualityQuery(fileId, repId)`	Returns current replica quality.

Figure 2: Quality monitor API.

replica becomes aware of concurrent updates that the former's value does not currently reflect (either because they have not yet been received, or because they are conflicting with the local tentative value), *qmon* is also notified (function *concurrentUpdatesKnown*).

As long as local volume managers provide the above input through the API, it is straightforward for *qmon* to calculate the quality vectors (along the 5 dimensions that Section 2 describes) for each replica it monitors.

It is out of the scope of this paper to describe how existing state-of-the-art replicated systems can call the above API functions to keep *qmon* up-to-date. Nevertheless, it is easy to show that the API is generic enough to fit a large class of replicated systems, both optimistic and pessimistic. Moreover, files from non-replicated file systems can also be included into the scope of *qmon*: such volumes simply need to register their files as single-replica files, no calls to other functions in the API are needed in this case.

3.1 Exposing replica quality to users

A 5-dimension notion of replica quality is not user-friendly. In order to be easily understood by regular users, we linearize the 5-dimension vector to a much more readable representation as a scalar value.

Different formulae can be used to linearize the 5-dimension vector. In the current version, we consider a simple formula where each dimension is first converted to a binary value. Value 0 means that the indicator for the given dimension is above some threshold, pre-defined for that dimension, while value 1 means the opposite. For instance, regarding the dimension *number of local tentative updates*, replicas with more local tentative updates than a pre-configured threshold (e.g. 3) will have a 0 value on that dimension, or 1 otherwise. Having converted each dimension to a binary value, we then sum each such component and normalize the corresponding result in order to obtain a scalar that may range from 0% (meaning very poor replica quality) to 100% (meaning excellent replica quality).

Such a scalar value can then be presented to the user in a graphically intuitive fashion, by converting each value to a range of colors (e.g. from red, meaning 0%, to green, meaning 100%), or to other intuitive symbols. Figure 3 illustrates a possible way to integrate quality feedback on typical graphical user interfaces.

For instance, in directory browsers, the icon representing a given replica in the current directory can be enriched with a symbol denoting its current quality (Figure 3). The interested user may then obtain more detailed information about the causes of such a color value by explicitly soliciting the system for further details (e.g. by asking for a *File Properties* window).

The notion of replica quality can also be applied to working sets, i.e., sets of files logically related, either by belonging to the same user, being included in a project (e.g., a book), having been manipulated by the same application (i.e., a task working set), or simply by location (e.g., a directory). Naturally, the replica quality of the working set must take into account the quality of each individual replica. However, they must be aggregated into a single value that represents the entire working set.

The user can choose different quality-of-service enforcement to working-set replica quality by deciding how aggregate values are calculated. The most demanding would be the aggregate quality being the minimum quality of the comprising replicas. This has the effect of devaluing an entire working set because of a single replica with lower quality and, therefore, the system will be continuingly attempting to improve the quality of replicas with lowers quality.

A more best-effort approach would be to consider an arithmetic average of replicas' quality. This way, a few replicas with lower quality will not hinder significantly the aggregate quality of, e.g., a directory. An intermediate approach can be devised by employing geometric average of replicas' quality as the aggregate working-set quality. This promotes working-sets comprised of replicas with more evenly balanced individual replica quality.

In task-based systems where the working set of a task is known (beforehand or based on use) [7, 8], the replica quality of members of the working set can be aggregated to represent the replica quality of the whole task.

The example in Figure 3 follows the same approach that popular graphical clients of remote revision control systems (e.g. Tortoise CVS [4]) already employ to present some status about local replicas. In practice, such an approach is possible in most graphical operating systems by having *qmon* intercept operating system events such as when file opening and closing, and windows focus changes.

We believe that providing users with such a continuously up-to-date quality feedback can substantially improve their experience when accessing replicated data. Empirical evidence from collaborative applications shows that users typically have multiple pending tasks to work on [5]. This means that, considering a given time period, the working set of a given user is actually dictated by her (possibly arbitrary) choice of which tasks she decided to carry out in that period.

This way, when facing the choice between two or more tasks to pursue at a given moment, each of which involving a distinct set of replicas, possibly with distinct qualities, such a feedback will encourage the user to select tasks which will manipulate fresher data and which will most likely have their work successfully committed.

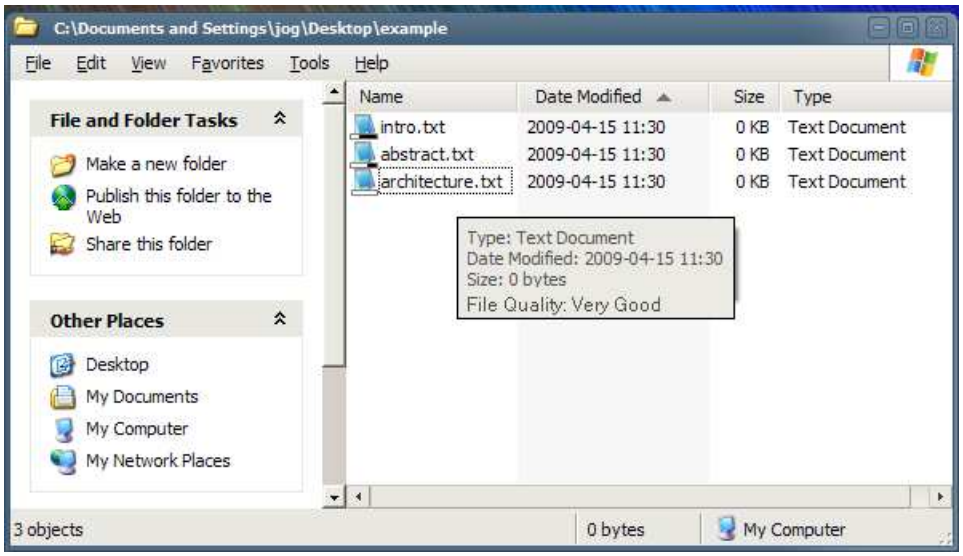

Figure 3: Illustration of *qmon* integrated with Windows Explorer. The user is encouraged to work on replica *architecture.txt*, which currently has excellent quality, and delay its work on other replicas until they recover an acceptable quality level.

4. REGULATING CONNECTIVITY

So far, we have seen how the system can improve user experience by providing it with an estimate of the quality of the replicas she accesses. The user can then react to such estimate, in at least two ways. When the user is about to opt between more than one task, he can filter out those that currently have unacceptable working set quality estimates. Perhaps more interestingly, if the user finds that no task has acceptable working set quality, they she can increase connectivity (possibly paying a monetary or energy price) until she obtains the desired quality.

When the second case is system-driven, we call it data-aware connectivity regulation. An easy way to define data-aware connectivity regulation is by analogy to a thermostat. Here, rather than a target temperature, the user sets up a desired minimum level of quality for a set of replicas of interest. Accordingly, the system automatically tries to continuously satisfy such a request by dynamically enabling only a set of available connections that ensures the former with minimal cost.

The following sections start by describing the basic mechanism to regulate data-aware connectivity, and then discuss how to improve it with replica access prediction techniques.

4.1 Basic data-aware connectivity regulation

Data-aware connectivity regulation is handled by a *q-regulator* daemon process, as illustrated in Figure 1. By interaction with the operating system's network management services, this component continuously keeps track of (i) the network adapters installed on the local device and, for each adapter that is currently enabled, (ii) the available connections that are available from that adapter.

Associated with each individual adapter and each connection, *q-regulator* maintains a bi-dimensional *cost vector*. Its first value denotes the current battery cost of the adapter or connection, while the second value corresponds to its monetary cost. For each connection, its *effective cost vector* is given by the sum of the connection's cost vector and the cost vector of the adapter through which the connection is established. Normally, cost vectors associated with adapters will have a null monetary cost, and a positive battery cost; whereas connections will have null battery cost, and null or positive monetary costs.

As an example, consider the case of a user carrying a laptop that enters the range of a paid WIFI hot-spot. The laptop's WIFI adapter will, for example, have a cost vector of $\langle 10, 0\rangle$, whereas the connection to the hot-spot will have a cost vector of $\langle 0, 25\rangle$. Since actually connecting to the hot-spot requires using the WIFI adapter, hence spending the corresponding battery power, and paying for each connection minute, the effective cost vector will be $\langle 10, 25\rangle$.

Since the operating system does not normally provide (or even know about) information on cost vectors, *q-regulator* relies on the user to explicitly provide such information. More precisely, the first time *q-regulator* detects an adapter or a connection, it allows the user to input the corresponding cost vector. In the absence of information from the use, *q-regulator* assumes a conservative default cost vector, configurable by the user.

As a future direction, it would be interesting to devise means of automatically obtaining cost information. Not only would this relieve the user from such task, but it could also allow more accurate values. For instance, most wireless adapters consume variable battery power, depending on variables such as signal range and the transmission error rate (determined by variable environmental factors such as noise in the room where the mobile device is). If cost information was provided by the adapter, rather than by the user, *q-regulator* would be able to instantaneously and accurately learn about each fluctuation in battery cost as the environment changes.

The user controls *q-regulator* via two parameters. A first one is *target replica quality*, a value denoting the minimum quality value that the user expects from any replica that any

Algorithm 1 Two-phase connectivity upgrade algorithm.

```
1: Let curCost be the effective cost vector of the currently active
   connections.
2: /* Connection Phase */
3: for all Installed adapter, adp, such that cost(adp) + curCost <
   maxCost do
4:   if adp is disabled then
5:     Enable adp.
6:     curCost = curCost + cost(adp).
7:     for all Connection, con, available through adp such that
       cost(con) + curCost < maxCost do
8:       Connect con.
9:       curCost = curCost + cost(con).
10:      if quality(file to open)< target replica quality then
11:        Break.
12:      end if
13:    end for
14:  end if
15: end for
16: if quality(file to open)< target replica quality then
17:   Close opened connections and enabled adapters.
18:   Return error.
19: end if
20: /* Disconnection Phase */
21: for all Active connection, actCon, in non-increasing order of
    effective cost do
22:   Disconnect actCon.
23:   if quality(open file set)< target replica quality then
24:     Reconnect actCon.
25:   else
26:     curCost = curCost − cost(actCon).
27:     if Adapter, adp, of actCon has no other active connections
        then
28:       Disable adt.
29:       curCost = curCost − cost(adp).
30:     end if
31:   end if
32: end for
```

application opens. A second parameter is *maximum cost*, the maximum vector cost that the user lets *q-regulator* spend to achieve the target replica quality. Essentially, the responsibility of *q-regulator* is to enable as few connections and adapters as possible (whose aggregate effective cost must be lower than the maximum cost parameter) such that, for the current set of open files of user, their quality is at higher or equal to the target replica quality.

Hence, besides information about the available adapters and connections (and corresponding cost vectors) and the above parameters, *q-regulator* keeps track of the set of open files of the current user. *q-regulator* monitors such a set by intercepting *openFile* and *closeFile* calls to the local file systems' API. When *q-regulator* learns of a newly opened replica (i.e. when it intercepts a call to some *openFile* function), it queries *qmon* for information about such a replica. If *qmon* has a registered entry about the replica and its current quality value satisfies the target replica quality parameter, then *q-regulator* lets the intercepted call to *openFile* complete normally.

If otherwise, *q-regulator* determines that the file to open has insufficient quality, it triggers an algorithm that tries to upgrade connectivity until the desired quality is reached. A possible solution is to follow the two-phase approach in Algorithm 1.

The two-phase connectivity upgrade algorithm starts by eagerly opening inactive connections (and, if necessary, enabling the corresponding adapter), one by one, until it obtains enough connectivity to ensures the intended quality value for the replica being opened (lines 3 to 15).

Of course, when opening every available connection is not sufficient to ensure the algorithm's goal, it returns to the original connectivity state and returns an error (line 18. *q-regulator* redirects such an error to the application that called the *openFile* whose interception triggered the connectivity upgrade algorithm, thus not allowing the replica to be opened.

The user can react to such an error in two ways. The user can increase the maximum cost parameter and retry opening the replica, hoping that the system will now be able to obtain enough connectivity to satisfy target replica quality. If, otherwise, the user cannot afford a higher maximum cost, it can simply disable *q-regulator* (either for that replica only or globally), therefore intentionally assuming the risk of accessing a poor quality replica.

If, otherwise, one or more connections that the first phase opened accomplished the target replica quality, the second phase tries to close superfluous connections (lines 21 to 32). This includes connections that were opened but do not actually contribute to improve replica quality of the current set of open replicas. The above algorithm follows a simple trial-and-error approach, which disconnects each connection and checks whether the aggregate quality of open replicas has decreased below the target value. If so, it reconnects the former.

Since different combinations of connections can attain similar aggregate quality values, the algorithm tries to end up with the combination that yields the lowest overall effective cost. It tries to accomplish that goal in phase 2, by heuristically disconnecting the most costly connections first (line 21).

Both phases can incur considerable delays, as well as battery and monetary overheads, since their trial-and-error nature may imply connecting and disconnecting frequently. As a direction for future work is to study alternatives to such an approach. A possibility is to devise intelligent estimates of the attainable changes in replica quality that result from connecting or disconnecting a given connection, without actually having to perform such operations. Such estimates can be based on past experience with such a connection.

Of course, *q-regulator*'s action is not exclusive to the moment where the user opens new replicas. When replicas are closed, *q-regulator* runs phase 2 of Algorithm 1, trying to alleviate the device from the cost of connections that are no longer necessary to ensure quality of the smaller, remaining set of open replicas.

Furthermore, even in periods when the set of open replicas remains constant, their quality may fluctuate. Recalling Section 2, this can, for instance, occur when some remote replicas become inaccessible, or when the local device learns of updates issued by inaccessible remote users. *q-regulator* keeps track of such dynamic changes to the quality of open replicas by passing *qmon* a reference to a callback function that *qmon* should call when the quality of any open replica happens to change considerably. When such a situation occurs, *q-regulator* reruns Algorithm 1.

4.2 Predictive connectivity regulation

The concept of replica quality can be used not just for a posteriori decisions as discussed up to now, where users decide whether to use data based on data quality achieved beforehand. Replica quality can also be used to a priori to guide replication updates, if integrated with some mechanism that predicts the user's working set.

More precisely, existing techniques for automated hoarding [21, 15, 14, 11] can continuously supply predictions of the expected near-future working set of the user to *q-regulator*. In turn, *q-regulator* can then run its connectivity upgrade algorithm in order to ensure the target replica quality, not just concerning the currently open replica set, but also the predicted future working set. This way, in the cases where the prediction is correct, the delay of the connectivity upgrade algorithm is hidden by running it ahead of time.

A further step is to integrate our mechanisms with calendar information. Calendar information feeds useful information into connection prediction. *q-regulator* can label locations that their calendars regularly refer with their connectivity properties. Once a reasonably large set of characterized locations is stored, *q-regulator* can infer future connectivity properties from the locations of future calendar appointments. Furthermore, for sparsely populated periods of future calendars, this mechanism can be enhanced by extrapolating past periodic user behavior to the future.

q-regulator takes a configurable amount of past calendar history, and for each future calendar slot calculates the most likely event based on past periodic occurrences (daily, weekly, monthly), thereby creating a tentative map of future activity.

5. RELATED WORK

Replication has been employed in distributed systems numerous times to increase availability, resorting both to pessimisticas well as optimistic approaches [19]. Pessimistic approaches based on locking has been traditionally used in distributed file systems and database transaction processing. Optimistic approaches have been applied originally to file systems and cooperative work applications in mobile scenarios. They aim essentially at providing increased availability during disconnection periods.

A relevant class of optimistic approaches that includes our proposed mechanisms are those that try to enforce some kind of guarantees on replica quality and therefore on user-experience for application operation (i.e., ensure a minimum of consistency or bound inconsistency or divergence) beyond plain eventual consistency such as offered, e.g., by Bayou [22], which are discussed next. Such inconsistency bounding approaches have been employed in web replication, database transaction processing, and distributed games in order to increase availability and reduce network bandwidth usage. When these bounds are about to be exceeded regarding a given replica, further updates on the replica are forbidden or delayed until the replica is made consistent.

The amount of time an object replica is allowed to remain stale, and still be accessed without it being required to be refreshed, can be bound by a maximum value in [1], therefore specifying a real-time guarantee on replica quality. A similar bound on the number of uncommitted updates allowed to be applied to a replica is used in order bounding [12], so that a transaction may be executed faster while being allowed to dismiss the effects of a bounded number of other preceding transactions.

TACT [23] integrates the two previous notions with an additional criteria concerning numeric error, i.e., by bounding the maximum value difference that may be produced on each replica by applying updates to it, in essence, allotting an update *quota* to each replica. Numeric error can be traced back to escrow techniques [13] employed in mobile databases during disconnection periods to ensure later transaction commitment on reconnection with servers. The three criteria are combined in the first multidimensional consistency enforcement model described in the literature.

In distributed gaming scenarios, while the strict consistency offered by pessimistic approaches is desirable, the communication costs make it a prohibitive option. To enforce this on large game scenarios, with dozens/hundreds of servers and several thousands players would simply bring the whole system to a halt or, at the very least, hinder game-play severely. Therefore, many distributed games employ variants of pessimistic approaches combined with information regarding players' position within the game scenario. The interest management [17] (or locality-awareness) hypothesis states that players are more interested (and are more subject to influence from) in what takes place on a limited radius around their current position. This simplification, somewhat artificially, limits the radius of players' interaction with others and is taken into account in game design, rules, scenario, and message routing protocols.

In vector-field consistency [20], optimistic replication, divergence bounding and locality-awareness have been combined not to overcome disconnection (as full connectivity among servers is mostly implied) but to provide better scaling and game experience. Hence, VFC enables greater (essentially unlimited) interaction radius, while reducing network bandwidth usage. VFC allows game developers or administrators to specify for each relevant game entity (player, enemy, inventory object, team flag, etc.) an approximation of a vector-field (by analogy with an electrical or gravitational field) that maps a consistency vector (containing consistency degrees) to each position (e.g., a 3D vector) in space (i.e., game scenario). The consistency field specifies bound values on time, sequence, and value, inspired partially on TACT and previous work. Bounds on time and sequence refer to replica staleness and missing updates, while bounds value refer to the magnitude (in percentage) of replica divergence produced by earlier updates and/or difference (in percentage) to game-related values (game winning score, kills left, distance to team flag, having a specific object in one's inventory, etc.). The vector-field is approximated by concentric circular (or rectangular for faster implementation) consistency zones or regions of increasing radius (or side size), and also increasing bounds defined in the consistency degrees. VFC provides graceful degradation in data consistency perceived by each replica as the distance from player's position increases, instead of the all-or-nothing approach of enforcing strict consistency to a limited radius employed in other game middleware [3, 2]. It provides a multidimensional consistency model inspired by TACT while embodying notions of (spatial) locality-awareness relevant in gaming scenarios.

Discussing our proposed mechanisms with the related work presented, both TACT and VFC aim at capturing a notion of replica quality, albeit indirectly and with different goals. Existing work ensures replica quality by essentially forbidding access to replicas once the divergence bounds have been exceeded and until they are obeyed again. In our work, we have a stronger goal: when replica quality is below the bounds defined, we attempt to regulate available connectivity in order to enforce those bounds. Once all connections are explored and this goal still unattained, only then we forbid replica access. Moreover, neither TACT nor VFC try to

estimate replica quality in a user-perceivable manner, nor they include the notion of replica quality applied to working sets or tasks.

6. CONCLUSIONS

We introduce the concept of data-aware connectivity to relieve users of having to pay high networking costs or of having to manually manage network connections in mobile settings in order to adjust the quality of replicated data.

As discussed, network data quality can be monitored and adjusted automatically via network connection management while complying with declarative networking cost restrictions. The *qmon/q-regulator* architecture integrates well with existing operating systems and replicated data infrastructures since it only monitors data access operations (*qmon*) and executes simple on/off operations on network connections (*q-regulator*). This leads to acceptable quality of accessed replicated data at minimal connectivity cost.

As future work, we plan to implement a prototype of the mechanisms introduced in the paper, and to use them in personal devices running relevant replicated systems. Such a prototype will enable measuring the overheads of this additional layer, regarding performance, network consumption and memory requirements.

7. REFERENCES

[1] R. Alonso, D. Barbara, and H. Garcia-Molina. Data caching issues in an information retrieval system. *ACM Transactions on Database Systems (TODS)*, 15(3):359–384, 1990.

[2] R. Balan, M. Ebling, P. Castro, and A. Misra. Matrix: Adaptive middleware for distributed multiplayer games. *ACM/IFIP Middleware Conference*, 2005.

[3] J. Chen, B. Wu, M. Delap, B. Knutsson, H. Lu, and C. Amza. Locality aware dynamic load management for massively multiplayer games. *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 289–300, 2005.

[4] T. CVS. Tortoisecvs. http://www.tortoisecvs.org/, 2009.

[5] M. Czerwinski, E. Horvitz, and S. Wilhite. A diary study of task switching and interruptions. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 175–182. ACM New York, NY, USA, 2004.

[6] P. J. Denning. The locality principle. *Commun. ACM*, 48(7):19–24, 2005.

[7] A. Dragunov, T. Dietterich, K. Johnsrude, M. McLaughlin, L. Li, and J. Herlocker. TaskTracer: a desktop environment to support multi-tasking knowledge workers. In *Proceedings of the 10th international conference on Intelligent user interfaces*, pages 75–82. ACM New York, NY, USA, 2005.

[8] J. Garcia and P. Ferreira. Operating system support for task-aware applications. In *Conference on Mobile and Ubiquitous Systems*, June 2006.

[9] F. P. Junqueira and K. Marzullo. Coterie availability in sites. In *DISC*, pages 3–17, 2005.

[10] P. Keleher. Decentralized replicated-object protocols. In *Proc. of the 18th Annual ACM Symp. on Principles of Distributed Computing (PODC'99)*, 1999.

[11] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 213–25. ACM SIGOPS, Oct. 1991.

[12] N. Krishnakumar and A. Bernstein. Bounded ignorance: a technique for increasing concurrency in a replicated system. *ACM Transactions on Database Systems (TODS)*, 19(4):586–625, 1994.

[13] N. Krishnakumar and R. Jain. Escrow techniques for mobile sales and inventory applications. *Wireless Networks*, 3(3):235–246, 1997.

[14] G. H. Kuenning, W. Ma, P. Reiher, and G. J. Popek. Simplifying automated hoarding methods. In *MSWiM '02: Proceedings of the 5th ACM international workshop on Modeling analysis and simulation of wireless and mobile systems*, pages 15–21, New York, NY, USA, 2002. ACM.

[15] G. H. Kuenning and G. J. Popek. Automated hoarding for mobile computers. In *Proceedings of the 16th Symposium on Operating Systems Principles*, pages 264–275, St. Malo, France, Oct. 1997. ACM.

[16] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28:690–691, September 1979.

[17] K. Morse et al. *Interest Management in Large-scale Distributed Simulations*. Information and Computer Science, University of California, Irvine, 1996.

[18] D. Peleg and A. Wool. The availability of quorum systems. *Information and Computation*, 123(2):210–223, 1995.

[19] Y. Saito and M. Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, 2005.

[20] N. Santos, L. Veiga, and P. Ferreira. Vector-field consistency for ad-hoc gaming. In *ACM/IFIP/Usenix International Middleware Conference (Middleware 2007)*, Lecture Notes in Computer Science. Springer, September 2007.

[21] C. D. Tait, H. Lei, S. Acharya, and H. Chang. Intelligent file hoarding for mobile computers. In *Mobile Computing and Networking*, pages 119–125, 1995.

[22] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the fifteenth ACM Symposium on Operating Systems Principles*, pages 172–182. ACM Press, 1995.

[23] H. Yu and A. Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of Operating Systems Design and Implementation*, pages 305–318, 2000.