The *Bancomat* Problem: An Example of Resource Allocation in a Partitionable Asynchronous System

Jeremy Sussman and Keith Marzullo

University of California, San Diego
Department of Computer Science and Engineering
La Jolla, CA 92093–0114
{jsussman,marzullo}@cs.ucsd.edu

Abstract. A partition-aware application is an application that can make progress in multiple connected components. In this paper, we examine a particular partition-aware application to evaluate the properties provided by different partitionable group membership protocols. The application we examine is a simple resource allocation problem that we call the *Bancomat* problem. We define a metric specific to this application, which we call the *cushion*, that captures the effects of the uncertainty of the global state caused from partitioning. We solve the *Bancomat* problem using three different approaches for building partition-aware applications. We compare the different group membership protocols in terms of the cushions they provide when solving the *Bancomat* problem.

1 Introduction

There exist several specifications and protocols for group membership in systems that can suffer partitions [13, 9, 2, 5]. Informally, there is a set of core properties that they all share, but they differ in the exact properties that they provide. These systems are meant to provide a basis for implementation of what has been called *partition-aware* applications, which are applications that are able to make progress in multiple concurrent partitions (that is, in multiple connected components) without blocking [2].

An essential problem confronted when building any distributed system is the uncertainty at any process of the global state. Partition-aware applications are especially sensitive to this problem because actions taken in one connected component cannot be detected by the processes outside of that component. Furthermore, when communication failures cause the system to partition, the processes may not agree at the point in the history that the partition occurred. The first issue must be directly addressed by the application, and partitionable group membership protocols help processes address the second issue.

In this paper, we examine a particular partition-aware application to evaluate the properties provided by different partitionable group membership protocols. The application we examine is a simple resource allocation problem that we call the *Bancomat* problem. We define a metric specific to this application, which we call the *cushion*, that captures the effects of the uncertainty of the global state caused from partitioning. The cushion is not the only interesting metric for the *Bancomat* problem, in that one cannot say that one partitionable group membership protocol is absolutely better than another because it allows a protocol to have a smaller cushion. Other metrics, such as message complexity, message size, and latency are important from a practical point of view. However, the cushion metric does give a measure of how well a given partitionable group membership protocol addresses uncertainty in the global state for the purposes of solving the *Bancomat* problem.

There are three main contributions of this paper. First, this paper specifies and examines a useful partition-aware application. We are not the first to consider this application, but we have not found a specification detailed enough to allow for a comparison of the properties of partitionable group membership protocols. Second, the paper contains such a comparison of partitionable group membership protocols. We believe that this comparison complements more taxonomic ones, such as [11]. Finally, the paper presents three different approaches to writing partition-aware applications: one in which no state is explicitly shared among the processes in the system and the processes take unilateral actions based on their local states; one in which all of the processes in a connected component share the same state and the actions are tightly coordinated in the component; and one in which processes in a connected component share state and a process informs the other processes when it has taken an action.

The paper proceeds as follows. Section 2 defines the *Bancomat* problem and the cushion metric, and gives a lower bound for the cushion metric. Section 3 presents the system model. Section 4 reviews the properties that the group membership protocols that we examine provide. Section 5 presents a solution for the *Bancomat* problem that uses a very weak idea of group membership to provide an upper bound on the cushion metric. Section 6 and Section 7 present two different solutions which require stronger group semantics, one based on a total delivery order and one without a total delivery order, and compare the cushions that result from building these solutions on top of the various group membership protocols. Section 8 summarizes our findings.

2 The *Bancomat* Problem and the *Cushion* Metric

The problem that we consider is loosely based on automatic teller machines, and so we call it the *Bancomat* problem.[1] This problem is a kind of resource allocation problem, where there is a relatively large number of identical resources that can be allocated. A practical example of such a service is a wide-area license service, where a relatively large but bounded number of clients can have a license to

[1] *Bancomat* is a common European term for an automatic teller machine. It doesn't suffer from the possible confusion that could arise if we were to name this the *ATM* problem.

use a software package simultaneously. Issuing licenses is implemented using withdrawals, and returning licenses is implemented using deposits.

There are a collection of n processes $A = \{b_1, b_2, \ldots b_n\}$ called *bancomats*. Collectively, the bancomats maintain a balance B of money, initially B_0. A client process can make two kinds of requests to a bancomat: it can ask for d amount of money to be withdrawn and it can ask for d amount of money to be deposited, where in both cases d is a (nonnegative) parameter of the request. When a client requests a bancomat to deposit d, d is added to B and the request terminates. When a client requests a bancomat to withdraw d, the bancomat can give the money to the client in pieces (deducting it from B as it goes), but the request will not terminate until d has been withdrawn.

We assume that bancomats are fully connected by asynchronous point-to-point channels: $c_{i,j}$ is the channel through which b_i sends messages to b_j. If b_i sends a message to b_j when $c_{i,j}$ is not faulty, then b_j will eventually deliver the message. If b_i sends this message when $c_{i,j}$ is faulty, then b_j will never deliver this message. We do not require that $c_{i,j}$ be faulty exactly when $c_{j,i}$ is faulty.

We define a *connected component* C of the bancomats A to be a maximal set of A for which $\forall\ b_i, b_j \in C : c_{i,j}$ and $c_{j,i}$ are not faulty. Each bancomat is in exactly one connected component at any time. The set of connected components can change over time, and components can split, can combine, and can simultaneously do both.

Given a bancomat b_i, we denote with d_i the amount of deposits that have been submitted, r_i the amount of withdrawal requested, and w_i the amount withdrawn at that bancomat. There are two safety properties: $\Box(w_i \leq r_i)$ and $\Box(\sum_{i=1}^{n}(w_i - d_i) \leq B_0)$. The first property states that money is not withdrawn from a bancomat unless it has been requested, and the second states that the balance is never negative.

We don't specify a liveness property for the protocol. We are interested in having solutions for systems that can suffer partitions, and specifying liveness properties for partitionable group membership has proven to be difficult. Instead, we define a metric that we call the *cushion*. Consider an infinite run in which there are no deposits and there have been withdrawal requests sufficient to receive as much money as can be withdrawn. The balance will eventually stabilize on some value, which we call the *final balance*. This balance represents money that the system did not allow to be withdrawn. The *cushion* is defined as the maximum final balance of any run in which every connected component received withdrawal requests for more than the initial balance B_0. A smaller cushion is desirable, since it ensures that the clients will be able to access more of their money.

Formally, consider a protocol Π that implements a bancomat system, and consider the set of runs $\rho \in R_\Pi \subseteq \Pi$ that satisfy the following constraints:

1. ρ is infinite.
2. There are no deposits.
3. Channels that fail remain failed.

4. For any connected component that persists forever, at least one bancomat in that connected component will receive at least B_0 withdrawal requests.

We define the cushion of the protocol Π to be the largest possible final balance: $cushion_\Pi = \max\ \rho \in R_\Pi : B_\rho$.

In any solution to the *Bancomat* problem for an asychronous system, messages are used to move money, explicitly or implicitly, both into or out of the system and among bancomats. In a partitionable system, any message that transfers money between bancomats is susceptible to the *Two Generals Problem* [10]—there is no way to ensure that both the sender and the receiver will agree on whether the last message sent between them has been successfully received. Since messages transmit money, the transmittor must assume, if given no reason not to, that the money was transferred even though it was not. That is, such money cannot be further used by the transmittor, since it cannot be certain that the receiver did not get the money. If the receiver did not receive the money, it cannot use it either. This money is unavailable for future transactions, and must be accounted for in the cushion. From this point forward, we will refer to this unavailable money as "lost".

In order to bound the amount of money that can be lost in a run, there must be a limit on both the number of messages that can suffer from this problem and the amount that can be lost by an individual message. The former can be addressed by only allowing there to be one message in transit from one bancomat to another, and the latter by specifying a *quantum* value of transfer. To fairly compare the different solutions and group membership protocols, we consider the quantum value a constant q, and impose the restriction of one outstanding message per process per group. A smaller quantum will allow for a smaller cushion, but will increase the message complexity. In this paper, we ignore this trade-off and only consider the cushion metric when comparing protocols.

3 Partitionable Group Membership System Model

We assume a system model that supports the partitionable group membership protocols that we consider. Processes communicate only through sending messages over a network. The network may partition. The exact details of what constitutes a partition have proven difficult to define, and so we rely on whatever abstraction each protocol provides. We do rely on one property that all protocols provide: if the network partitions the processes into a set of components, and then the processes remain in these components forever, then eventually the group membership protocols will detect this condition and form groups equal to these components. Section 4 discusses this property further.

We assume that the network preserves FIFO order, in that if process p sends message m to process p' and then sends message m' to process p', then p' may receive just m, or just m', or m before m', but never m' before m. Note that this is not assumed by the system models of the partitionable group membership protocols we consider, but most of the protocols we consider implement such a FIFO ordering.

Since the system model admits message loss, the Two General's Problem occurs with respect to bancomats adjusting the balance and clients withdrawing or receiving money. [10] We avoid this problem by defining correctness in terms of withdrawals and deposits that have been made at each bancomat.

If bancomats can crash, then the values d_i, r_i and w_i would need to be stored in stable storage and the balance would be defined in terms of the values recorded to stable storage. We avoid this additional complexity by assuming that bancomats do not crash.

Under this system model, and defining the cushion only in terms of runs in which failed links do not recover, one would expect that a reasonable protocol would not have a cushion larger than $n(n-1)q$. This is because no more than q can be lost by the failure of any channel. In [16], we show that a lower bound on the cushion is $\lfloor n/2\log(n)\rfloor q$.

4 Properties of Partitionable Group Membership Services

We consider six different partitionable group membership protocols:

1. Extended virtual synchrony communication (hereafter EVSC) [13] used by both the *Transis* [7] and *Totem* [1] systems;
2. A protocol that we call asynchronous virtually synchronous communication (hereafter AVSC) that is provided for application use by the *Transis* system; [6]
3. Weak virtually synchronous communication (hereafter WVSC) [9] used by the *Horus* system [15];
4. A protocol that we call Unibo (for the University of Bologna, where it was developed) that was designed specifically for wide-area network based applications [2] [2];
5. Two protocols associated with the specification given by Cristian and Schmuck in [5]. The specification does not include communication properties which are needed to solve the *Bancomat* problem. This specification was meant to be instantiated with one of two sets of communication properties [4]. We call the two resulting protocols CS1 and CS2 (for the initials of the last names of the developers).

Partitionable group memberships provide the abstraction of *teams* and *groups*. A team specifies an abstract set of processes that communicate with each other to provide a service, and a group is a concrete set of processes associated with a team. Processes associated with a team *install* a group, which provides the process with an identifier for the group and a set of process identifiers, called the *membership* of the group. In order to differentiate groups with the same membership, a unique identifier is associated with each group. A group is installed at most once by each process. Once a process installs a group, it is said

[2] An earlier version of the specification of this protocol can be found in [3]. In terms of cushions for the *Bancomat* application, the differences are irrelevant.

to be *in* that group until it installs another. If a process p installs a group g and then installs group g', we say that p *regroups from* g *to* g'. For the purposes of the *Bancomat* problem we need only one team that defines the abstract set of machines.[3]

One can impose a relation on groups based on their installation by a process: g precedes g' if a process p regroups from g to g'. All group membership protocols ensure that this relation is irreflexive and asymmetric. An irreflexive strict partial order can be defined by taking the transitive closure of this relation, and two groups that are not related by this order are said to be *concurrent*.

All of the protocols that we consider in this paper use the group installed at a process to approximate the component to which that process belongs. They differ in the tightness of this approximation. However, all share the property that if the system stabilizes into a permanent set of components, then each process will eventually install a group whose membership is the members of the component and will forever remain in that group.

We denote with $|g|$ the number of processes that are in group g. We say that a group g is *fully formed* when all processes in the membership of g install g. There may be groups that are never fully formed, and a process may not know when a group is fully formed. All protocols ensure that concurrent fully formed groups do not have overlapping memberships.

All protocols allow a process to broadcast a message m to a team of which it is a member. All processes that deliver m must be members of the team to which m was broadcast, and all must be in the same group when they deliver m. This group must contain the sender. However, not all members of the group may deliver the message.

There are two ways that a process can determine which processes received a message. One method is based on *message stability*. A message m is said to be stable within a group when all processes in the group have received m, and a message is stable at a process p when p knows that the message is stable in p's group. A message can become stable only after it has been delivered in a fully formed group. All protocols considered in this paper provide a mechanism for alerting the members of a group when a message becomes stable. Some of the protocols offer an option to not deliver a message until it becomes stable.

The second method for a process to learn which processes received a message is based on regrouping. Suppose a process p regroups from g to g'. Define the *survivor set* $SS(g, g')$ to be those processes that installed g, and then installed g' without installing any intermediate groups. All of the protocols guarantee that all of the members of $SS(g, g')$ have delivered the same set of messages while in group g. These members also agree on the stability of these messages: if p in $SS(g, g')$ knows that m became stable in g, then all of $SS(g, g')$ know that m became stable in g.

[3] Some protocols use the term *group* to indicate what we refer to here as a team, the term *view* to indicate a group, and the verb *to join* a group to indicate to install a group.

For each process, there is a point when it leaves group g and a later point when it joins the successor group g'. These two points define what we call the *regrouping interval* from g to g'. Group membership protocols differ in how messages are delivered and whether messages can be sent during regrouping intervals. In particular,

– EVSC does not allow messages to be sent during regrouping intervals. Outside of regrouping intervals, if the option to only deliver stable messages is chosen, then at the end of regrouping intervals a block of unstable messages may be delivered.
– CS1 does not allow messages to be sent during regrouping intervals. CS1 only delivers stable messages both outside and during regrouping intervals. Notification can be given when a message is known by all members of the group to be stable.
– WVSC allows messages to be sent during regrouping intervals, but these messages are delivered in the successor group g'. Messages can be delivered at all times, and need not be stable to be delivered.
– AVSC, CS2 and Unibo all allow messages to be sent during regrouping intervals. Their delivery semantics with respect to the stability of messages correspond to EVSC, CS1 and WVSC respectively.

Let g be the group that p has most recently installed when it sends a message m. All group membership protocols guarantee that m, if delivered, will be delivered in a group that does not precede nor is concurrent with g. Group membership protocols EVSC, CS1, WVSC and Unibo further restrict the group in which m is delivered. Specifically,

1. EVSC stipulates that m is delivered in g.
2. CS1 stipulates that if m is delivered, then it is delivered in g. A process p will not deliver m if p does not know that m is stable.
3. WVSC stipulates that if p sends m outside of a regrouping interval, then m will be delivered in g, and if p sends m during a regrouping interval, m will be delivered in the subsequent view g'.
   In addition, during a regrouping interval WVSC provides a sequence of zero or more membership lists $\langle V_1, V_2, \dots V_\ell \rangle$ that are all supersets of the membership of g'. These membership lists, which are called *suggested views* are nested: $V_1 \supseteq V_2 \supseteq \dots \supseteq V_\ell$. One could think of these membership lists as defining a sequence of groups, but none of these groups would become fully formed and no messages would be delivered in them. Hence, as is done in [9], we treat them simply as membership lists.
4. Unibo stipulates that if m is sent during a regrouping period, it will be delivered in the group g' that is installed after the regroup concludes. This, combined with the FIFO ordering that is also part of the Unibo specification, ensures that if m was not sent in a regrouping interval, m will either be delivered in g or in g', the next group installed by p.

The group membership protocols provide optional delivery order semantics. Most provide *causal ordering* options, in which messages delivered in the same group are delivered in a manner that respects the causal order defined in [12]. Also, many provide a *total order* option, in which all messages delivered in a group have been assigned a unique order in that group, and a message is delivered by a process only if all of the messages which preceed that message in the group have been delivered.

5 First Approach: No Shared State, Unilateral Actions

The first solution that we examine is formally presented in [2]. Informally, each bancomat maintains a local balance. The initial balance is initially partitioned in some manner among the bancomats. When a bancomat receives a withdrawal request, it will immediately fulfill the request without communication if there are sufficient funds in the local balance. If the local balance is insufficient to fulfill the request, then the bancomat requests a transfer of funds from some other bancomat. If this bancomat cannot transfer sufficient funds, then the original bancomat asks another bancomat for a transfer, and so forth. When the original bancomat receives sufficient funds to fulfill the request, it completes the transaction. Deposits are added to the local balance of the bancomat that receives the deposit request.

5.1 Cushion and Group Membership Requirements

The cushion for this protocol depends on the way a bancomat b_i, with insufficient funds, requests a transfer of funds from b_j. One strategy would have b_i not request a transfer from yet another bancomat b_k until either b_j responds with insufficient funds or b_i regroups into a group without b_j before receiving the response. The other strategy allows b_i to make concurrent transfer requests. With the latter strategy, the cushion is $n(n-1)q/2$. A run that attains this cushion is as follows. Let b_n be the first bancomat to receive withdrawal requests. When b_n has no more money in its local balance, it asks all the other bancomats for transfers. When each of the other bancomats send a quantum to b_n, the communication between that bancomat and b_n fails. This contributes $q(n-1)$ to the final balance. The next bancomat to receive requests, b_{n-1}, performs similarly, and the communication fails similarly, adding $q(n-2)$ to the final balance. Thus, there will be $\sum_{i=1}^{n}(i-1) = n(n-1)/2$ messages lost, resulting in a final balance of $n(n-1)q/2$. This is the worst possible final balance, since if a channel $c_{i,j}$ fails with money in transit, then b_j will never get from b_i a subsequent request for a transfer.

With the former strategy, the cushion is smaller but has the same complexity: as proven in [16] it is $\lfloor n/2 \rfloor \lceil n/2 \rceil q$. The run constructed above can not occur. Bancomat b_n lost the transfer from b_{n-1}, and so b_n sent the transfer request to b_{n-2} while in a group not containing b_{n-1}. Therefore, b_{n-2} must be in a group without b_{n-1} when it delivers the transfer request, and b_{n-1} will therefore not make the transfer request to b_{n-2}.

This solution requires very little from the group membership service. Indeed, for the larger cushion, all that is needed is a mechanism whereby a bancomat decides that another bancomat is disconnected. For the smaller cushion, all that is needed is the simple message delivery property that a message sent in group g is not delivered in a group preceeding or concurrent with g. All group membership protocols that we consider implement this property. Hence, using the weakest partitionable group membership protocol improves the cushion over using *no* protocol, but any further strengthing of group membership is not useful for this approach.

6 Second Approach: Shared State, Unilateral Actions

The second approach for the *Bancomat* problem has the optimal cushion of $n/2\log(n)q$. We give an informal description of the protocol here; the complete protocol is detailed in [16]. We build the protocol on top of a group membership protocol in which concurrent fully-formed groups are disjoint, messages are delivered in the group in which they are sent, and messages cannot be sent during regrouping intervals. This is equivaluent to EVSC.

The complexity of the protocol arises from withdrawals, and so we temporarily ignore deposits. A client sends a request for a withdrawal to a bancomat. The bancomat waits until it can safely issue a quantum of money to the client, then issues that money, and broadcasts this fact to its group. We say that a message m is *application stable* when all members of the group have delivered m. Once the bancomat knows that its withdrawal request is application stable, it repeats the process until either the request is satisfied or it can no longer safely issue a quantum.

A bancomat can safely issue a quantum when it knows that by doing so the group balance will remain nonnegative. It is possible for all other bancomats in its group to concurrently issue a quantum of money, and so it is safe for a bancomat to issue a quantum only when the group balance is at least the quantum value multiplied by the size of the group.

Suppose bancomat b regroups from group g to group g'. The bancomat computes a final value for the group balance of g, and then contributes its share of this balance towards the group balance of g'. We define the *final group balance* of group g to be the initial value of the group balance of g minus all quanta that were delivered in g. Unfortunately, b may not have delivered all of the withdrawal requests delivered in g, and so it must compute an upper bound on the number of quanta withdrawn.

Recall that a bancomat can send a withdrawal request only after the previous withdrawal request it sent has become application stable. Hence, b knows that each bancomat that was in g but is not in g' may have sent one withdrawal request that b did not deliver. The upper bound on the number of withdrawal requests sent in g is the number that b delivered plus one for each bancomat that left g. Note that this message must be application stable; message stability as defined in Section 4 is not sufficient. A message is application stable when the states of all of the processes in the group reflect that message. A message is stable when all processes in the group are guaranteed to eventually deliver the message or crash. This solution requires the former.

Let b' be a bancomat that left g. If at some later time b' joins a group containing b also, then b can tighten its estimate of the final group balance of g. It does so by b' telling b (using a *state transfer* message) how many quanta it withdrew while in g.

Hence, b computes the group balance for the new group g' as follows. It first computes its share of the final group balance of g. From the properties of the group membership protocol, all bancomats in $SS(g, g')$ compute the same share, and so b includes these shares into the group balance of g'. Then, for each bancomat b' in g' but not in g, b waits for a state transfer message from b' that contains b''s contribution to the group balance of g' and the number of quantum it delivered the last time it was in a group with b. If b installs yet another group without receiving this message from b', b computes the group balance of g without b''s contribution. Since b''s contribution is always nonnegative, omitting this contribution is always safe.[4]

Deposits are implemented as follows. A bancomat b quantizes the deposit amount. b broadcasts the first quantum deposit notification to the group. When b knows that the deposit notification is stable, it broadcasts the next quantum, and so on. Upon delivery of a deposit notification, each bancomat increases the group balance by a quantum. Those that do not deliver the deposit still have a safe estimate of the final group balance. These bancomats will learn of the deposit via a state transfer message if they eventually join a group which b also joins.

6.1 Cushion

In [16] we show that this protocol has a cushion of $\lfloor n \log(n)/2 \rfloor q$. Informally, the worst-case run is constructed as follows. For simplicity, we consider n to be a power of 2. In this run, each connected component of bancomats repeatedly splits into two equal-sized connected components. This continues until there are n connected components, each containing one bancomat. At this point, each connected component receives B_0 withdrawal requests.

When a connected component of size $2k$ splits into two components of size k, then the bancomats in one component must assume that each bancomats in the other component sent a withdrawal request that was lost due to the split. Hence, each component deducts kq from the final balance of the original connected component. Amortized per bancomat, each bancomat contributes $q/2$ to the cushion for each time that bancomat joins a new group. Each bancomat joins $\log(n)$ groups, and so the cushion is $nq \log(n)/2$.

[4] If the state transfer message is lost due to the failure of a communications channel, then the amount carried by this message will be added to the final balance. State transfer messages occur only when channels recover, and so runs with state transfers are not considered when the computing the cushion of a protocol. The money is still lost, however. In practice, state transfers should also be quantized.

6.2 Group Membership Requirements

This protocol requires that a message be delivered in the group in which it was sent, and that concurrent fully-formed groups be disjoint. In addition, the protocol was written with no messages sent during regrouping intervals. These are the properties that are provided by EVSC, and so this protocol can be run, as is, on EVSC. CS1 also provides these properties, but in CS1 a message will not be delivered if it does not become stable. Thus, when a bancomat b sends a message m in g, b must rebroadcast m in the subsequent group g' should b not deliver m in g. This does not affect the cushion.

Unlike EVSC and CS1, WVSC allows for messages to be sent during regrouping intervals. A simple way to port the protocol to WVSC is for a bancomat to not send any messages during a regrouping interval, to ignore all suggested views, and to perform the actions that occur due to a regroup event at the end of the regrouping interval. One can modify the protocol, however, to allow bancomats to send messages (in particular, withdrawals) during regrouping intervals.

To do so, b computes a conservative estimate of the initial balance of g': b assumes that it is the only bancomat that brings any funds to the new group. In addition, the current suggested view is a superset of the membership of g'. For b to allow a withdrawal to occur during a regrouping interval, it ensures that its (conservative) share of the conservative balance is sufficient to cover the withdrawal. If so, b sends the withdrawal request; otherwise, b waits for the regrouping interval to complete.

In both cases, no additional messages can be lost over the original protocol, and so the cushion for both versions of the protocol on EVSC have the same optimal cushion as before. The second WVSC protocol may perform better than the original protocol because withdrawal requests are not automatically blocked during regrouping intervals. Since regrouping uses timeouts and is usually based on multiple rounds of communication, the performance improvement may be significant.

Adapting this protocol to run on top of AVSC, CS2, and Unibo is harder because a sending process knows very little about the group in which its message will be delivered. As with WVSC, we use a conservative approach. Before a bancomat b sends a withdrawal request, it first computes a conservative initial group balance for a hypothetical group in which the withdrawal request might be delivered. In order for this group balance to be conservative, b assumes that this group arose by having b first regroup into a group by itself, all bancomats except for b reduce their balances to zero, and then all the bancomats join a group with b. Bancomat b sends a withdrawal request only if this conservative balance is sufficiently large. Thus, if b is in a group of size k and that has a group balance of B, then it can withdraw a quantum only when $(B/k)/n \geq q$.

This protocol has a cushion that is at least $q(n^2 - 1)$. Consider the run in which all bancomats remain connected and all withdrawal requests are sent to b_n. It will continue to allow withdrawals through $B = qn^2$. Once the final quantum is taken, b_n will allow no more withdrawals giving a final balance of $q(n^2 - 1)$.

This is a very conservative protocol, and it is an open question whether there is a less conservative version.

7 Third Approach: Shared State, Coordinated Actions

The third approach has the bancomats in a group share their state. This is provided by totally-ordered group multicast, with stable message notification, as described in Section 4. The protocol is given in [16], and is informally described here.

The protocol is similar to the one of Section 6. The main difference is in how withdrawals and deposits are handled. As before, requests are broken into quanta and handled sequentially. In the earlier protocol, a bancomat will allow a withdrawal of a quantum if its share of the group balance is at least a quantum. In this protocol, a bancomat first broadcasts the request to withdraw a quantum to the team, and does not check for sufficient funds until it delivers this request. For this protocol, "sufficient funds" means that the group balance is at least a quantum. Thus, in this protocol, withdrawal requests can be rejected due to insufficient funds even when the reqesting bancomat had sufficient funds in its local balance when it did the broadcast.

Since the requests are delivered in a total order, each bancomat that delivers a request r will agree on the group balance when r is delivered, and will therefore take the same action. Bancomats other than the sender b of the request r can act on r as soon as they deliver it, but b must wait to act until r becomes stable in the group. By waiting until r becomes stable, b guarantees that all other members of its group will include r in any final balance they compute for the group.

Rebalancing is similar to the protocol of Section 6. The only difference is in the computation of the final balance of a group. Consider a bancomat b in $SS(g, g')$. In the previous protocol, b assumes that any bancomat not in $SS(g, g')$ had sent a withdrawal notification in g that b did not deliver. Hence, b includes such possible notifications when computing the final balance for g. In the protocol of this section, b knows that any withdrawal request from a bancomat b' not in $SS(g, g')$ must be stable at b' before b' performs the withdrawal. Thus, b includes a withdrawal request from b' in the final balance of g only when it has delivered such a request in g. As with the earlier protocol, this is a conservative estimate: r may never have become stable at b'.

7.1 Cushion and Group Membership Requirements

In [16] we show that this protocol has a cushion of $\lfloor n \log(n)/2 \rfloor q$. Informally, the worst-case run is the same as for the protocol of Section 6.

This solution requres the group membership service to provide total ordering of messages and stability notification. All of the protocols that we examine in this paper can supply both. Since these are the only requirements needed for this solution, the stronger group membership protocols may provide more than is needed. Indeed, a protocol such as that suggested in [8] is sufficient for this solution. Total ordering comes at a cost, however, especially in a wide-area network.

8 Discussion

In this paper, we (1) examine a partition-aware problem, (2) discuss three different approaches to solving partition-aware problems, and (3) compare how well different group membership protocols support solutions to this problem. In this section, we make some observations and raise some questions about these three issues.

8.1 Partition-Aware Problems

We were surprised at how hard it was to find a partition-aware problem that was concrete enough to be amenable to formalization. For example, [2] lists four different partition-aware problems, one of which is a version of the *Bancomat* problem. We have tried to formalize the other three, but so far have had only limited success in defining a metric, like the cushion, that captures the value of the properties of different partitionable group membership protocols.

We suspect that there are only a few different kinds of partition-aware applications. If it is the case, then it might be worthwhile to design partitionable group membership protocols with these specific applications in mind.

8.2 Different Approaches for Partition-Aware Problems

We examined three different approaches to solving partition-aware problems: one in which processes act autonomously and communicate as infrequently and with as few processes as possible, one that generalizes the state machine approach [14] to partitionable systems, and one that is an intermediate approach; processes act autonomously but broadcast their actions to their connected component.

The first approach is appealing because it uses very little from the group membership service. We were surprised that one property about message delivery in groups was sufficient to lower the cushion from $qn(n-1)/2$ to $q\lfloor n/2\rfloor\lceil n/2\rceil$. The required property does not appear to be very expensive to provide.

The state-machine-like approach also does not require much from the group membership service, but what it does require is not cheap: total message delivery order within a connected component. A totally-ordered multicast is required before every withdrawal, which implies that the latency for this protocol could be high.

The intermediate approach strikes a balance between these two, but we don't yet know the value of such a balance. Our suspicion is that it should perform better, but we have not yet tested this hypothesis.

8.3 Group Membership Protocols and the *Bancomat* Problem

The differences between the different group membership protocols were most important for the intermediate approach of Section 6. Using a weak partitionable group membership protocol like AVSC, CS2 and Unibo resulted in a large cushion, while the other protocols allow for an optimal cushion. On the other hand, the protocol for the weak membership services is extremely conservative. We are currently trying to design a less conservative version.

It has been suggested that there are a class of applications that require the EVSC-supplied property that a message is delivered in the group in which it was sent. This class of application has been named *group aware* [6]. The *Bancomat* problem is not group aware by this definition, but we suspect that without either at least the WVSC delivery properties or a total ordering on message delivery, it cannot be solved with an optimal cushion.

Our experience with this problem has led us to reconsider how partitionable group membership services should be presented. Many of the differences appear to be irrelevant with respect to implementing at least this partition-aware problem. Instead of concentrating on providing different properties, it might be worthwhile to provide more information to the application concerning the state of the system when communication fails. The fundamental problem we had to confront when designing these protocols was bounding the possible states of the processes in different connected components. Having more information might allow one to further restrict the possible states.

Acknowledgements We would like to thank Özalp Babaoğlu, Karan Bhatia, Idit Keidar, Alberto Montressor and Aleta Ricciardi for their help with this paper.

References

1. Amir, Y.; Moser, L.E.; Melliar-Smith, P.M.; Agarwal, D.A.; et al. The Totem single-ring ordering and membership protocol. In *ACM Transaction on Computer Systems*, 13(4):311-42, November 1995.
2. Ö. Babaoğlu, R. Davoli, A. Montresor and R. Segala. System Support for Partition-Aware Network Applications. Technical Report UBLCS-97-08, Department of Computer Science, University of Bologna, September 1996. A shorter version of this paper appears in: *Operating Systems Review*, 32(1):41-56, January 1998.
3. Ö. Babaoğlu, R. Davoli and A. Montresor. Group Membership and View Synchrony in Partitionable Asynchronous Distributed Systems: Specifications. Technical Report UBLCS-95-18, Department of Computer Science, University of Bologna, September 1996.
4. F. Cristian. Personal communication.
5. F. Cristian and F. Schmuck. Agreeing on processor-group membership in asynchronous distributed systems. Technical Report CSE95-428, UCSD, 1995. Available via anonymous ftp at cs.ucsd.edu as /pub/team/asyncmembership.ps.Z.
6. D. Dolev. Personal communication.

7. D. Dolev, and D. Malki. The Transis approach to high availability cluster communication. In *Communications of the ACM*, vol. 39, (no.4), ACM, April 1996. pages 64–70.
8. A. Fekete, N. A. Lynch, and A. A. Shvartsman. Specifying and using a partitionable group communications service. In *Proceedings of The Sixteenth Conference on Principles of Distributed Computing*, Santa Barbara, CA, 21–24 August 1997, pages 53–62.
9. R. Friedman and R. van Renesse. Strong and weak virtual synchrony in Horus. In *Proceedings 15th Symposium on Reliable Distributed Systems*, Nigara-on-the-Lake, Ont., Canada, 23–25 October 1996, pages 140–9.
10. J. N. Gray. Notes on Data Base Operating Systems. In *Operating Systems: An Advanced Course*, Springer-Verlag Lecture Notes in Computer Science 60:393–481, 1978.
11. M. A. Hiltunen and R. D. Schlichting. Properties of membership services. In *Proceedings of the Second International Symposium on Autonomous Decentralized Systems*, Phoenix, Az., 25–27 April 1995, pages 200–207.
12. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, July, 1978, pages 558-565.
13. L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended virtual synchrony. In *Proceedings of the 14th International Conference on Distributed Computing Systems*, Pozman, Poland, 21–24 June 1994, pages 56–65.
14. F. B. Schneider. Implementing fault tolerant services using the state machine approach: A tutorial. In *Computing Services*, vol. 22, (no.4), December 1990. pages 299-319.
15. R. van Renesse, K. P. Birman, S. Maffeis. Horus: a flexible group communication system. In *Communications of the ACM*, vol. 39, (no.4), ACM, April 1996. pages 76–83.
16. J. Sussman and K. Marzullo. The *Bancomat* Problem: An Example of Resource Allocation in a Partitionable Asynchronous System. Technical Report CS98-570, Department of Computer Science and Engineering, University of California, San Diego, (revised) July 1998.