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

Jeremy Sussman
IBM TJ Watson Research Center
Hawthorne, NY

Keith Marzullo
University of California, San Diego
Department of Computer Science and Engineering
La Jolla, CA

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 four different approaches for building partition-aware applications. We compare the approaches in terms of their cushions and discuss how well different group membership protocols support these approaches.

1 Introduction

There exist several specifications and protocols for group membership in systems that can suffer partitions [20, 15, 2, 9]. 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 measures how much of the resource a distributed allocation service can hold back to ensure that it never overallocates. The solutions we give use different properties of partitionable group membership protocols. Thus, indirectly, the cushion of a solution also gives a measure of how well a given partitionable group membership protocol addresses uncertainty in the global state.

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 approach complements more taxonomic comparisons, such as [18]. Finally, the paper presents four different approaches to writing a partition-aware application. These approaches differ in the amount of coordination among the bancomats with respect to withdrawals and deposits. The four approaches are as follows:

1. Actions are serialized among the processes to provide tight coordination among them. We show that this approach has an optimal cushion.

2. No state is explicitly shared among the processes in the system and the processes take unilateral actions based on their local states. We show that the cushion must be larger in this case, but we don't show this bound to be tight. We show how using group membership halves the cushion of our solution.

3. All processes in a connected component share the same state and the actions are tightly coordinated in the component. We show that this approach has a low cushion.

4. Processes in a connected component share state and a process informs the other processes when it has taken an action. We show that this approach has the same low cushion as the previous approach but uses different properties of group membership.

The paper proceeds as follows. Section 2 defines informally the *Bancomat* problem and the cushion metric. Section 3 presents the system model. Section 4 gives a more formal definition of the problem and the cushion metric, and gives a lower bound for the cushion metric given the system model. Section 5 reviews the properties that the group membership protocols that we examine provide. Sections 6 through 9 present the four solutions described above, and compare the cushions that result from building these solutions on top of the various group membership protocols. Section 10 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] We give here an informal and incomplete description of the problem to introduce the key concepts; a complete specification is in Section 4.

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 simultaneously have licenses to use a software package.

There is a collection of n processes called *bancomats*. Collectively, the bancomats maintain a balance B of money that has an initial value of B_0. The balance is represented as an integer. 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

[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.

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 increments (deducting it from B as it goes), but the request will not terminate until d has been withdrawn. Deposits and withdrawals are expressed in integer multiples of a fixed quantum q, which is described below.

A safety property of the protocol is that B equals the initial balance B_0 plus the amount of money deposited minus the amount of money withdrawn. The solution must also satisfy the safety property that B is always nonnegative.

Ideally, the action of debiting an account of the amount d occurs if and only if the bancomat issues d. But, since messages can be lost, the Two General's Problem [16] applies to these actions and they cannot be guaranteed to occur atomically. We avoid this problem by defining correctness only in terms of debits and credits that have been made at each bancomat: the balance B is equal to the initial balance B_0 plus, for each bancomat b, the number of deposits b has credited minus the number of withdrawals that b has debited.

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.

As is shown below in Theorem 4.1, any interesting solution to the Bancomat problem requires bancomats to communicate with each other. Such communication is 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—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 transmitter must assume, if given no reason not to, that the money was transferred even when it was not. Such money cannot be further used by the transmitter, 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". (Such lost money can be recovered if communications is re-established).

Tto bound the amount of money that can be lost in a run, there needs 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.

3 System Model

We assume a system model that supports the partitionable group membership protocols that we consider. We assume an asynchronous distributed system where processes communicate only through sending messages over a network. The network may partition, resulting in *components*. A component is a set of processes such that all the processes in a component can communicate with each other but not with any process not in their component. The set of components changes over time and processes can arbitrarily move from one component to another or form new components. 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 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 5 discusses this property further.

We assume that communications provides FIFO delivery order: 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. For the others, it is simple to implement (usually using sequence numbers).

If bancomats can crash, then they would need to be store state on stable storage and the balance would be defined in terms of the state on stable storage. We avoid this additional complexity by assuming that bancomats do not crash.

4 Formal Specification and Lower Bound

Consider a system of n bancomats $A = \{1, 2, \dots n\}$. The bancomats together implement an account whose balance is denoted by B. We denote with B_0 the initial value of the balance, which can differ for different executions of the protocol. Withdrawals and deposits are done with a granularity of q. Given a bancomat i, d_i denotes the amount of deposits that have been submitted, r_i the amount of withdrawal requested, and w_i the amount withdrawn at i.

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 do not specify a liveness property for this problem. Instead, we define a performance metric that we call the *cushion*. This metric measures how much money might become unavailable for withdrawals due to communication failures. To compute the cushion, one considers all executions (or *runs*) of the protocol in which there are no deposits, in which all communication failures persist and in which each connected component receives a sequence of withdrawal requests totaling at least as much as the initial balance. By having no deposits, only the initial balance is available for withdrawing, and by having persistent communications failures any uncertainty introduced by a partition can not be resolved. The money that remains unwithdrawn in such a run represents money the system held back to ensure that the safety property is not violated.

More formally, consider a protocol Π that implements a bancomat system, and consider the set of runs R_Π of Π that satisfy the following constraints:

1. The run 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 a sequence of withdrawal requests that total to at least B_0.

5. The group membership protocol only suspects faulty channels as having failed.

For runs that satisfy these five constraints, the balance is monotonically decreasing. Given that the initial balance is finite, that the balance must remain nonnegative and that the balance decreases in multiples of q, the balance must eventually reach a final nonnegative value. We denote the final value a balance stabilizes on in such a run ρ as B_ρ. We call the *cushion* of the protocol Π the largest possible final balance: cushion of $\Pi = \max\ \rho \in R_\Pi : B_\rho$.

One would expect that a reasonable protocol would not have a cushion larger than $n(n-1)q$. This is because no more than a quantum q can be lost by the failure of any channel.

The fifth constraint listed above may seem surprising, since in an asynchronous system one cannot ensure that the only suspicions are accurate. However, without this constraint the group membership protocol can, in effect, force processes to communicate in a point-to-point manner by having all groups have only size two. In this case, the group membership protocol doesn't add anything over what is provided by simple point-to-point message delivery. As we show in Section 7, such a protocol has an $O(n^2q)$ cushion. Thus, without this constraint it is hard to compare different approaches to solving the *Bancomat* problem. In Section 10 we mention a possible way to make such a comparison without the last constraint.

4.1 Lower Bounds on Cushion

The lower bound on cushions depends on how bancomats communicate with each other to allow withdrawals. We present two lower bounds and describe the conditions on communication under which they are tight. In both cases, we do so by constructing a run that satisfies the five conditions given above and computing its final balance.

For any bancomat protocol, a bancomat that receives a sequence of withdrawal requests will eventually be unable to satisfy a request without receiving a message from another bancomat. This would occur, for example, if the initial balance was partitioned among the bancomats and the sequence of requests a bancomat received totaling more than its initial portion of the balance. Or, it would occur with the first request if the protocol requires all bancomats to reach some kind of agreement about each withdrawal request. We denote with u_i the amount of money that a bancomat can withdraw without receiving a message from another bancomat. Clearly, for any bancomat protocol there is a value u_i for each bancomat i where $0 \le u_i \le B_0$ and $\sum_{i=1}^{n} u_i \le B_0$.

Theorem 4.1 *Consider a protocol Π in which at least two bancomats do not allow w_i to become larger than u_i. Under these conditions the cushion is at least $B_0/2$.*

Proof: Let U be the set of bancomats i that do not allow w_i to become larger than u_i, and let bancomat j be the bancomat in U with the smallest value of u_j. Consider a run ρ in which there are no communication failures, and let only bancomat j receive withdrawal requests. Hence, eventually $r_j \ge B_0$ will hold. This bancomat will withdraw only u_j units, leaving a balance of $B_0 - u_j$. Since bancomat j has the smallest u_j in U and $\sum_{i \in U} u_i \le B_0$, $u_j \le B_0/|U|$. Thus, $B_\rho \ge (|U|-1)B_0/|U|$. By definition, $|U| > 1$ and the cushion must be at least as large as B_ρ. ■

Because of this fact, we restrict our discussion to protocols for which at least $n-1$ of the bancomats i will attempt to withdraw more than u_i. In doing so, we will never construct a run that has bancomat 1 wait for a message to withdraw more than u_1.

We refer to a bancomat i as being *drained* when $w_i = u_i$ and $r_i > w_i$ (i.e., it has given out u_i quanta and has requests to give out more). From the definition of the cushion metric, we have the freedom to construct runs in which any number of bancomats become (at least momentarily) drained. We also assume that we can have the initial values of u_i for all i be large enough to construct the runs in the proofs.

By definition of u_i, bancomat i needs to receive some message from another bancomat to withdraw more than u_i. We call such a message an *authorization message*. It could be a point-to-point message or a message multicast to a group of bancomats that includes i. Informally, an authorization message is a transfer of funds: if i emits a quantum having received an authorization message, then that quantum can't be emitted by any other bancomat. A communications failure, though, might keep i from receiving the authorization message. If i partitions from the rest of the bancomats, then a simple Two-Generals argument shows that the other bancomats can't determine whether i received the authorization message or not. If i in fact did not, then the quantum is effectively added to the final balance. Of course, there can be runs in which communications with i is re-established allowing the quantum to be recovered.

Theorem 4.2 *Consider a protocol* Π *that solves the* Bancomat *problem. There is a run* $\rho \in R_\Pi$ *that has a final balance* $B_\rho \geq (n-1)q$.

Proof: We construct such a run. The only constraints we have on this run are the five given above.

First send withdrawal requests totaling to at least $u_n + 1$ to bancomat n. To satisfy the last request n needs to receive an authorization message. Suppose that one is sent, and then partition n from the rest such that this message is lost. This represents one quantum that cannot be emitted.

Send send withdrawal requests totaling to at least $u_{n-1} + 1$ to $n - 1$. To satisfy the last request $n - 1$ needs to receive an authorization message. Suppose that one is sent, and then partition $n - 1$ from the rest such that this message is lost. This represents another quantum that cannot be emitted.

This process can be repeated for the rest of the bancomats. We can then have each (now isolated) bancomat i be sent withdrawal requests totaling to at least $B_0 - u_i - 1$, none of which can be satisfied since each is isolated and can therefore receive no authorization messages. The resulting state has a final balance of at least $(n-1)q$. ■

Corollary 4.1 *The lower bound of cushions for protocols that solve the* Bancomat *problem is* $\Omega(nq)$.

In Section 6 we present a *Bancomat* protocol that has a cushion of $O(nq)$, and so the lower bound from Corollary 4.1 is tight.

5 Properties of Partitionable Group Membership Services

We consider six different partitionable group membership protocols:

1. Extended virtual synchrony communication (hereafter EVSC) [20] used by both the *Transis* [12] 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; [11]

3. Weak virtually synchronous communication (hereafter WVSC) [15] used by the *Horus* system [25];

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 [9]. 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 [8]. We call the two resulting protocols CS1 and CS2 (for the initials of the last names of the developers).

We do not consider the question of the implementability of group membership in an asynchronous system [7]. Rather, we assume that the above protocols provide their advertised semantics assuming the expected operating environment. We then compare the impact of their different semantics on our different approaches to solving the *Bancomat* problem. We also don't consider the point that one can provide the semantics of one group membership service on top of another group membership service. Instead, our comparison is based on using each group membership service in a natural way. In doing so, our goal is to illuminate at least some of the effects of the choices the designers made when specifying their group membership protocols.

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 to be *in* that group until it installs another. If a process p installs a group g and then installs group g' without installing an intermediate group, 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 partial order 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 the transitive closure of this relation is irreflexive and asymmetric: if a process regroups from g to g', then it does so only once and no process regroups from g' to g. 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 the process 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 have installed 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

[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 between the two versions are irrelevant.

[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.

be in the same group when they deliver m. This group must contain the sender. However, some members of the group may not 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 regrouped from g to g'. 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.

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* for that process 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 a process to send a message during a regrouping interval. Outside of regrouping intervals, if the option to only deliver stable messages is chosen, then at the end of regrouping intervals a process may deliver a block of messages that it does not know to be stable.

- CS1 does not allow a process to send a message during a regrouping interval. Processes only deliver 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 a process to send messages during a regrouping interval, 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 processes to send messages during a regrouping interval. 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 [15], 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 [19]. 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 precede that message in the group have been delivered.

6 First Approach: No Concurrency

The first solution that we examine does not allow for concurrent withdrawals within a connected component. There are many ways that such a solution can be implemented. In the implementation presented here, we assume the basic properties of group membership are provided: that there are agreed upon groups installed at the group members, and that a message is delivered in the same group at any processes to which it is delivered. In this solution, the bancomats do not keep a local balance. Rather, they pool their money into a *group balance*. We describe below how the group balance is calculated.

When a group is installed, some deterministic algorithm is used to order the members of the group. The first bancomat in the order is elected the *token holder*. The token holder is the only bancomat that is permitted to take an action: it can withdraw a quantum from the group balance or deposit a quantum to the group balance if it has outstanding requests to do so from a client. If it has an action to perform, the bancomat broadcasts a message to the rest of the group informing them of the change in the group balance. When this message is stable, the bancomat withdraws or deposits a quantum. After performing the action, or if it has no action to take, the token holder then passes the token to the next bancomat in the order, and this bancomat becomes the token holder.

The group balance is calculated in a conservative manner to ensure that the safety properties described in Section 4 are not violated. When a group is split, a bancomat calculates the minimal possible group balance based on the messages that it has received in the group, e.g., if a token holder sent a message that a quantum was to be withdrawn, then the bancomat must assume that this quantum was withdrawn. Then, each bancomat divides the group balance evenly among the bancomats in the group, and takes its share to the next group. When a group is formed, all of the bancomats send a state transfer message that includes the balance that it brings to the group. The first token holder waits until it has received all of these state transfer messages and for these messages to be stable before taking its first action. Thus, all of the bancomats share the same initial group balance when withdrawals and deposits occur.

The protocol shown in Figure 1 is written in pseudocode as a set of routines that are invoked when certain events of interest occur. For the cushions described in this paper, only withdrawals and group splits are relevant and so we only present the pseudocode for these events. Those interested in the entire pseudocode can find it in [24]. The events of interest are a client of a bancomat requesting a withdrawal of k quanta (`ClientWithdrawal`), delivery of bancomat-generated messages (`Receive`), the installation of a new group (`Regroup`) and the notification that a message has become stable (`Stable`). In addition, there is a subroutine `ActAsTokenHolder` that is called by the bancomat when it is the token holder. The variable `b` signifies the local bancomat that is running the protocol.

6.1 Cushion

The cushion for this protocol meets the lower bound for cushions established in Corollary 4.1. The token in this protocol is used for this serialization. An authorization message can only be sent by a bancomat that holds the token, and the token is passed only after the authorization message becomes stable. Thus, there can be at most one authorization message in transit per group.

The implications of this use of the token directly lead to the cushion being $O(nq)$. When a regrouping occurs, there can be at most one message that is not stable in the group. Thus, there can be only one quantum that is lost due to message loss during regrouping. Since there can be at most $n-1$ splits in a run, the final balance must be less than nq.

6.2 Group Membership Requirements

This solution requires only the very basic properties of group membership. The group membership service is used to elect a token holder at the beginning of a group, to provide an agreed path for the token to follow in the group, and to detect the loss of a token via a message or process failure. These three functions all follow directly from the basic properties of group installation.

It is possible to achieve a similar cushion without the token abstraction. Consider for example a solution in which the balance is kept locally at a single bancomat, and any bancomat that wishes to perform a withdrawal or deposit does so only with the permission of the bancomat keeping the balance. A quantum would be lost only upon a message being lost between this bancomat and any of the $n-1$ others, providing a cushion of $(n-1)q$. Such a solution does not meet the requirements of a partition-aware application, since only the component containing the bancomat keeping the balance would make progress. We believe that any partition-aware solution that achieves a cushion linear in n will use some means similar to the token to allow only one withdrawal or deposit at a time. Thus, any such solution will show a similar requirement for a group membership protocol as shown here.

7 Second Approach: No Shared State, Unilateral Actions

The previous solution does not allow multiple bancomats to send authorization message concurrently. If we wish to allow such concurrent authorization messages, though, then the $\Omega(nq)$ lower bound on the cushion is not attainable.

Theorem 7.1 *Consider a protocol Π that solves the* Bancomat *problem where for all i, $u_i > \log n$ and where authorization messages are sent as point-to-point messages and can be sent concurrently. There is a run $\rho \in R_\Pi$ that has a final balance $B_\rho \geq \lfloor n \log n/2 \rfloor q$.*

```
 // Local group and balance for this bancomat
 Set group = initial group;
 int balance = initial balance;

 // keep track of requests from clients that are not fulfilled
 int requestedWithdrawals = 0;

 // identifier of current token holder
 ID TokenHolder;
--------------------------------------------------------------------------
ClientWithdraw(int k)
    requestedWithdrawals = requestedWithdrawals + k * q;

Regroup(Set newGroup, Set survivors)
    balance = balance * (|survivors| / |group|);
    group = newGroup;

    TokenHolder = first bancomat in group;
    if (TokenHolder = b) call ActAcTokenHolder;

Receive(message m from b′)
    if (m = ⟨Withdrawal⟩)
        balance = balance - q;
    else if (m = ⟨Token⟩)
        TokenHolder = bancomat after TokenHolder in group;
        if (TokenHolder = b)
            call ActAsTokenHolder;

ActAsTokenHolder()
    if (requestedWithdrawals > 0)
        send(⟨Withdrawal⟩);
    else
        send(⟨Token⟩);

Stable(⟨Withdrawal⟩ from b)
    requestedWithdrawals = requestedWithdrawals - q;
    output(q);
    if (TokenHolder = b)
        call ActAsTokenHolder;
```

Figure 1: *Bancomat* Protocol: No Concurrency

Proof: Construct ρ as follows: the n bancomats are initially in one connected component. Let D be a set of $\lfloor n/2 \rfloor$ of these bancomats and L be the remaining $\lceil n/2 \rceil$ bancomats. Send withdrawal requests to the bancomats in D until are drained and are waiting for an authorization message. For each bancomat i in D, choose a bancomat i' in L as the source of an authorization message destined for b_i.

Partition the bancomats into two sets of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ such that each set contains half of the drained bancomats and for each drained bancomat i the source of its authorization message i' is in the other set. Have this partition cause all authorization messages to become lost. Since all of the sources of authorization messages are uncertain as to whether their authorization message was delivered (and none were), $\lfloor n/2 \rfloor q$ is added to the final balance.

This procedure can be repeated: pairing each drained bancomat i with an undrained bancomat i' in the same partition, and losing all of the authorization messages due to a partition that halves the size of each of the two sets, thereby adding $\lfloor n/2 \rfloor q$ to the final balance. This procedure can be repeated until each of the drained bancomats end up in their own partition. This can be done $\lceil \log n \rceil$ times, resulting in a final balace of at least $\lfloor n \log n/2 \rfloor q$. ■

We now present a simple concurrent solution to the Bancomat problem. This solution was informally presented in [2]. Each bancomat maintains a local balance. The initial balance is initially arbitrarily partitioned among the bancomats. When a bancomat receives a withdrawal request for k quanta, it immediately fulfills 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.

The relevant parts of the protocol is shown in Figure 2.

7.1 Cushion

The cushion for this protocol depends on the way a bancomat i with insufficient funds requests a transfer of funds from bancomat j. We consider two strategies. The first strategy allows a bancomat to send concurrent transfer requests.

To compute the cushion of this protocol, consider a graph that has a node for each bancomat and a directed edge from i to j if i has sent an authorization message to j that j will never deliver. We call this the *cushion graph.* This graph depicts a global state of the protocol, and an edge in this graph represents a message that contributes one quantum to the final balance.

Recall that a cushion is computed only from runs in which there are no deposits. Thus, from the protocol, once a bancomat becomes drained it will never send an authorization message. A path from i to j and then to k in the cushion graph indicates that k became drained before j. Hence, the cushion graph contains no cycles. In addition, a lost authorization message results only from channel failures, and since we only consider runs in which failures persist there can not be multiple edges from one node to another. Thus, the final balance can therefore be no larger than $n(n-1)q/2$, which is the maximum number of edges one can have in an acyclic directed graph of n nodes. In the following theorem we construct a run with this final balance, thereby showing that this protocol has a cushion of $n(n-1)q/2$.

Theorem 7.2 *The protocol of Figure 2 has a cushion of* $n(n-1)q/2$ *when a single bancomat can make concurrent transfer requests.*

```
 // Local group and balance for this bancomat
 Set group = initial group;
 int balance = initial balance;
---------------------------------------------------------------------------
ClientWithdraw(int k)
    int issue = 0;
    if ( balance > 0 )
        issue  = min(balance, k * q);
        balance = balance - issue;
        output(issue);
    while ( issue < k * q )
        // choose a bancomat b′ from which to request funds
        send(⟨Request q⟩ to b′);
        wait(Receive(m from b′) ∨ (decides that cannot communicate with b′));
        if (m = ⟨Transfer of q⟩)
            issue = issue + q;
            output(q)

Receive(⟨Request q⟩ from b′)
    if (balance ≥ q)
        balance = balance - q;
        send(⟨Transfer of q⟩ to b′);
    else
        send(⟨Insufficient funds⟩ to b′);
```

Figure 2: *Bancomat* Protocol: No Shared State, Unilateral Actions

Proof: Start with the n bancomats in one connected component. Send requests to bancomat n until it is drained, and let it send transfer requests to the other bancomats. Partition bancomat n away from the rest losing the $n-1$ authorization messages. Continue by sending withdrawal requests to bancomat $n-1$ until it is drained and have it send transfer requests to the bancomats $1, 2, \ldots n-2$. Partition bancomat $n-1$ away from the rest losing the $n-2$ authorization messages. Continue to do so until each bancomat is in its own connected component. Send enough withdrawal requests to the bancomats to satisfy the requirements on the run being one from which a final balance can be computed. The final balance will be $q[(n-1)+(n-2)+\ldots+1] = n(n-1)q/2$. ■

This strategy uses *no* properties of the group membership protocol except that a bancomat sends messages only to bancomats in its own group. The second strategy instead uses the failure detection and regrouping properties of group membership. When a drained bancomat b sends a transfer request to b', it multicasting the request to its group (which contains both b and b'). b does not send a transfer request to another bancomat unless b' sends an insufficient funds message or b regroups to a new fully formed group that does not contain b'. And, for every authorization message am that b' sends to b (again by multicasting it to the group), b' will not send any transfer requests until either am is stable or b' regroups into a fully formed group without b.

With this strategy, the cushion graph will not only be acyclic, but also there will be no more than one (directed) path between any pair of nodes. We call such graphs *single-path DAGs.*

Lemma 7.1 *The cushion graph generated by the second strategy is a single-path DAG.*

Proof: Assume otherwise. Let b and b' be nodes such that there is more than one path from b to b' and, without loss of generality, where b has out degree of more than one and b' has in degree of more than one. Let two of the parents of b' be x and y. Assume that b first sent a transfer request to x, and x replied with an authorization message that was lost. From the second strategy, b' subsequently formed a group g that excluded x before it sent a transfer request to y. This group must include b because there is a path from b via y to b': the children of b will send their transfer requests to b and so must be in a group with b, and in the runs we consider for computing cushions, any regrouping results in a groups whose membership is a contained in the previous group.

Either x is b (in which case y is not b) or x is not b. If x is b, then g must both contain b (because there is a path via y from b to b') and exclude b (since otherwise it would not have lost the authorization message from x). If x is not b, then x will regroup to a new group g' before it sends a transfer request. The groups g and g' are concurrent and fully formed, and so can not have overlapping memberships. Since there are paths from b to x and b to y, though, both groups must contain b. ■

The following theorem establishes the maximum number of edges in a single-path DAG. We then construct a run that loses this number of authorization messages thereby establishing the cushion for the protocol.

Theorem 7.3 *A single-path DAG has no more than $\lfloor n/2 \rfloor \lceil n/2 \rceil$ edges.*

Proof: Consider a bipartite directed graph with the left side having d nodes, the right side having $n-d$ nodes, and with an edge connecting each node on the left to each node on the right. This graph is a single-path DAG. In addition, any path in this graph has a length of one, and any other edge added to this graph will either create a cycle or create two paths between a pair of nodes. The graph has $d(n-d)$ nodes, which is maximized when $d = \lfloor n/2 \rfloor$ or $d = \lceil n/2 \rceil$.

Consider some single-path DAG. If all paths have lengths of one, then it is a bipartite graph. Assume that there is at least one path that has a length $\ell > 1$. We show that we can redraw the graph so that the path has a length of $\ell - 1$, the number of edges in the graph is not reduced, and no cycles or multiple paths between two nodes are introduced. One can then repeatedly apply this redrawing rule to a graph until it is a bipartite graph. Thus, the maximum number of edges occurs in the bipartite graph given above.

The graph is redrawn as follows. Consider a maximal path that has a length $\ell > 1$. Choose some node b in this path that has an in degree $in > 0$ and an out degree $out > 0$. Remove b from the path and draw an edge from each parent of b to each child of b. Then, draw an edge from b to each sink in the graph (since it is acyclic, there must be at least one sink). Note that this will not introduce any cycles or any multiple paths. It will introduce some length one paths from b to the sink, and replace the original path of length ℓ with $in * out$ paths of length $\ell - 1$. It also removes $in + out$ edges and adds $in * out + s$ edges where s is the number of sinks. Since $in \geq 1, out \geq 1$ and $s \geq 1$ $in + out \leq in * out + s$. ■

Theorem 7.4 *The protocol of Figure 2 has a cushion of $\lfloor n/2 \rfloor \lceil n/2 \rceil q$ when the second strategy is used.*

Proof: Consider the directed bipartite cushion graph with bancomats $1 \ldots \lfloor n/2 \rfloor$ on the left side and $\lfloor n/2 \rfloor + 1 \ldots n$ on the right side and with edges from all the left side nodes to all of the right side nodes. This graph has $\lfloor n/2 \rfloor \lceil n/2 \rceil$ edges. We construct a legal run that constructs this cushion graph. In doing so, we have a process install a new group only when required to satisfy group membership requirements. We give the membership of the groups by listing the bancomats that must be excluded to satisfy group membership requirements.

All bancomats are initially in the same group with each other. Send requests to bancomat $d = \lfloor n/2 \rfloor + 1$ until it is drained. Have d send a transfer request to bancomat 1 and then disconnect d from 1 losing the authorization message. d joins a group that excludes 1. d then sends a transfer request to bancomat 2, disconnects from 2, and so on through bancomat $\lfloor n/2 \rfloor$ losing authorization messages along the way . At this point,

- bancomats 1 and $\lfloor n/2 \rfloor + 2$ through n are still in the initial group containing all bancomats;
- each bancomat i for $1 < i \leq \lfloor n/2 \rfloor$ is in a group that excludes 1 through $i - 1$;
- bancomat $\lfloor n/2 \rfloor + 1$ is in a group that contains only itself.

We now have bancomat $d' = \lfloor n/2 \rfloor + 2$ follow the same path that d did. When d' delivers its own transfer request destined for bancomat 2, d' will join 2's group. Joining this group will ripple down the left hand side bancomats as d' repeatedly sends transfer requests. At the end,

- bancomats 1 and $\lfloor n/2 \rfloor + 3$ through n are still in the initial group containing all bancomats;
- each bancomat i for $1 \leq i \leq \lfloor n/2 \rfloor$ are in a new group that excludes only 1 through $i - 1$;
- $\lfloor n/2 \rfloor + 1$ and $\lfloor n/2 \rfloor + 1$ are in groups that contains only themselves.

This procedure is applied to the remaining bancomats $\lfloor n/2 \rfloor + 3$ through n. ■

7.2 Group Membership Requirements

This solution requires very little from the group membership service. Indeed, the properties assumed in Theorem 7.2 requires *no* partitionable group membership service! The resulting cushion, however, is as large as one would expect it could be for any reasonable protocol. The properties assumed in Theorem 7.4 allow one to reduce the cushion, but not reduce the asymptotic complexity.

8 Third Approach: Shared State, Unilateral Actions

The third approach for the *Bancomat* problem is a concurrent solution with a cushion of $\lfloor n \log n/2 \rfloor q$. We first give an informal description of the protocol and then give the complete protocol in 8.1. We build the protocol on top of a group membership protocol equivalent to EVSC: 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.

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. Once the bancomat knows that every other bancomat in its group is aware of this withdrawal, it repeats the process until either the request is satisfied or it can no longer safely issue a quantum. We say that a message is *application stable* when each bancomat in the group has delivered the message to its application and the application has processed the message. Hence, a bancomat i can issue the next quantum of money only after it knows that its previous withdrawal notification is application stable.

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. This is why a bancomat b in group g waits for its prior withdrawal notification to be application stable before it withdraws another quantum. It is only at that point that b knows all of the bancomats b' in g have included its prior withdrawal into their views of the group balance.

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 issued in g. Unfortunately, b may not have delivered all of the withdrawal notifications sent in g, and so it needs to compute an upper bound on the number of quanta withdrawn.

Recall that a bancomat can send a withdrawal notification only after the previous withdrawal notification 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 notification that b did not deliver. The upper bound on the number of withdrawal notification sent in g is the number that b delivered plus one for each bancomat that left g.

Let b' be a bancomat that left g. If at some later time b' joins a group that contains b, then b can tighten its estimate of the final group balance of g. It does so by having b' tell 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.

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.

8.1 Formal Description

Figure 3 shows the relevant pseudocode for the the protocol. The event handler `ClientWithdraw` handles clients withdrawal requests for k quanta. As described in Section 2, to minimize the cushion only one money transfer can be sent by a bancomat to a group at any time. Therefore, this event handler updates the local state to reflect that a withdrawal request was received from a client, and then the subroutine `TryToSendQuantum` is called to determine when a quantum can actually be withdrawn. Withdrawals are not given to the client until they are reflected in the group balance.

The event handler `Regroup` is called when a new group has been installed. The relevant parts shown here include calculating the balance and then, since a regroup implies that there are no outstanding withdrawal notifications in the new group, an attempt to send a withdrawal notification can be made.

The subroutine `TryToSendQuantum` is called by the event handlers to determine when a withdrawal request can be issued by `b`. In this protocol, a withdrawal notification can only be sent when all previously sent withdrawal notification are *application* stable and the group balance will not become negative if all bancomats concurrently request a quantum. The protocol uses the toggle `sentQuantum` to determine whether there is a withdrawal notification sent by `b` that has not become application stable. If this is not the case, then the group balance is divided by the members of the group and compared to the quantum value to see if the withdrawal notification can be sent.

The two event handlers `Receive` and `ApplicationStable` are called when a withdrawal notification is received or becomes application stable respectively. The withdrawal notification change the group balance, and possibly allow a pending withdrawal request to be sent.

8.2 Cushion

In Theorem 8.1 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 (we argue later why this is the worst case). 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 bancomat in the other component sent a withdrawal notification 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

```
  // Local group and balance for this bancomat
 Set group = initial group;
 int balance = initial balance;

// keep track of requests from clients, and whether have outstanding messages
 bool sentQuantum = false;
 int requestedWithdrawals = 0;
--------------------------------------------------------------------------------
ClientWithdraw(int k)
    requestedWithdrawals = requestedWithdrawals + k * q;
    TryToSendQuantum();

Regroup(Set newGroup, Set survivors)
    balance = (balance - (|group - survivors| * q)) * (|survivors| / |group|);
    group = newGroup;
    sentQuantum = false;
    TryToSendQuantum();

TryToSendQuantum()
    int share = (balance/|group|);
    int actualWith = min(requestedWithdrawals, q);
    if (¬sentQuantum && ((share > actualWith) && (actualWith > 0)))
        sentQuantum = true;
        requestedWithdrawals = requestedWithdrawals - actualWith;
        send(⟨Withdrawal of actualWith⟩);
        TryToSendQuantum();

Receive(⟨Withdrawal of w⟩ from b′)
    balance = balance - w;
    if (b′ = b)
        output(w);
    send(⟨acknowledge m⟩ to b′);

ApplicationStable(⟨Withdrawal of w⟩ from b)
    sentQuantum = false;
    TryToSendQuantum();
```

Figure 3: *Bancomat* Protocol with Some Shared State, Unilateral Actions

the cushion for each time that bancomat joins a new group. Each bancomat joins $\log n$ groups, and so the cushion is $(n \log n/2)q$. We first prove a lemma necessary for Theorem 8.1.

Lemma 8.1 *Consider the recurrence relation*

$$f(g) = g + \max_{1<k\leq g} \max_{\text{all possible } g_i} \sum_{i=1}^{k} \left(f(g_i) - \frac{g_i^2}{g} \right) \quad \textit{where } \forall i : 1 \leq i \leq k : g_i > 0 \quad \textit{and} \quad \sum_{i=1}^{k} g_i = g$$

where $f(1) = 0$ and $f(2) = 1$. The function that solves this relation is $f(g) = g \log(g)/2$.

Proof: To prove that this function does solve the recurrence relation, we first show it assuming that the maximum occurs for $k = 2$ and $g_1 = g_2 = g/2$, and then show that these values give the maximal solution.

Substitute the function $f(g) = g \log(g)/2$ into the original formula.

$$f(g) = g + \max_{1<k\leq g} \max_{\text{all possible } g_i} \sum_{i=1}^{k} \left(g_i \log(g_i)/2 - \frac{g_i^2}{g} \right) \quad \text{where } \forall i : 1 \leq i \leq k : g_i > 0 \quad \text{and} \quad \sum_{i=1}^{k} g_i = g$$

To find the maximum of this function over k and the g_i values for any given g, we need to maximize the summation. Assuming f is maximized when $k = 2$ and the two g_i values are $g_1 = g_2 = g/2$, we get

$$f(g) = g + g_1 \log(g_1)/2 - g_1^2/g + g_2 \log(g_2)/2 - g_2^2/g$$

$$= g + 2 \cdot (g/2 \cdot \log(g/2)/2 - (g/2)^2/g)$$

$$= g + g \log(g/2)/2 - g/2$$

$$= g/2 + g(\log(g) - 1)/2 = g \log(g)/2$$

Thus the maximized function is in fact the recurrence relation.

The second part of this requires showing that the summation is maximized when $k = 2$ and $g_1 = g_2 = g/2$. We prove this in two steps.

1. For $k = 2$ we show that any non-zero, non-negative choices of g_1 and g_2 such that $g_1 + g_2 = g$ give a sum that is smaller than when $g_1 = g_2 = g/2$.

2. For any value of $k > 2$, the sum can be increased by decreasing the value of k. Since k must be at least 2, this proves by induction that the function is maximized when $k = 2$.

Thus, by these two steps, the function is shown to be maximized when $k = 2$ and $g_1 = g_2 = g/2$.

In Figure 4(a), we present the graph of $f'(x) = x \log(x)/2 - x^2/\ g$ (shown as a dark line). For comparison we include a line from the origin to the point $(g, f'(g))$. For further clarity, in Figure 4(b), we show the difference between the function and this line. The function $f'(x)$ has the following properties:

- The function f' crosses the comparison line at exactly three points, $x = 0$, $x = g$ and $x = g/2$. The function f' crosses the line at $x = 0$ and $x = g$ by the definition of the line. To show that the function crosses the line at $x = g/2$, we show that $f'(g/2) = f'(g)/2$.

$$f'(g/2) = (g/2) \log(g/2)/2 - (g/2)^2/g$$

$$= (g/4)(\log(g) - 1) - g/4$$

$$= g/4 \log(g) - (g^2/g)/2 = f'(g)/2$$

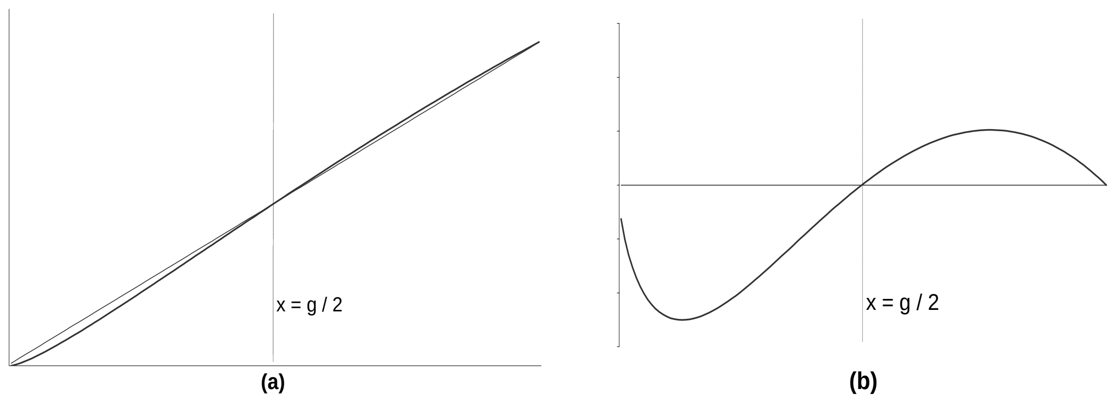

Figure 4: **(a)**$f'(x) = x\log(x)/2 - x^2/\ g$ from $0 < x \leq g$. **(b)**$f'(x) - (x \cdot f'(g)/g)$ from $0 < x \leq g$.

- For the interval $0 < x < g/2$ the function remains below the comparison line, and for the interval $g/2 < x < g$ the function remains above the line.
- The difference between the function and the comparison line in the first interval is larger than in the second interval: that is, for any $0 < x \leq g/2$, $f'(g - x) - (g - x)(f'(g)/g) \leq x(f'(g)/g) - f'(x)$.
- The graph is monotonically increasing, and from $1 \leq x < g/2$, the slope of the graph increases monotonically.

For part (1) of the proof, we set $k = 2$. Let g_1 be any point on the graph, $0 < x < g$. Since $\sum_{i=1}^{k} g_i = g$, if $g_1 = x$ then $g_2 = g - x$. One of these values must be less than or equal to $g/2$, so without loss of generality we can let $x \leq g/2$. Then, as shown above,

$$f'(g - x) - (g - x)(f'(g)/g) \leq x(f'(g)/g) - f'(x)$$

$$f'(x) + f'(g - x) \leq x(f'(g)/g) + (g - x)(f'(g)/g) = f'(g)$$

Since, as shown above, $f'(g/2) \cdot 2 = f'(g)$, for all other choices of x, $f'(x) + f'(g - x)$ is no greater. Therefore, this function is indeed maximized when $g_1 = g_2 = g/2$.

For part (2) of the proof, consider any choice of $k > 2$ values of g_i. Since all g_i are non-negative, and they sum to g, at most one of the g_i values can be greater than $g/2$. Therefore, there must be at least two g_i values that are less than $g/2$, which we will call g_{k-1} and g_k. Now, consider the choice of $k' = k - 1$ values $g_1 ... g_{k'}$, where $g_{k'} = g_{k-1} + g_k$. If $f'(g_{k'}) > f'(g_{k-1}) + f'(g_k)$, then the original function $f(g)$ must be increased by using k', i.e., by choosing a smaller k value. What remains to be shown is that $f'(g_{k'}) > f'(g_{k-1}) + f'(g_k)$.

If $g_{k'}$ is greater than $g/2$, then $f'(g_{k'})$ must be above the comparison line. However, both $f'(g_{k-1})$ and $f'(g_k)$ were below the line. So the sum of $f'(g_{k-1})$ and $f'(g_k)$ must also be below the line. Thus, $f'(g_{k-1} + g_k) > f'(g_{k-1}) + f'(g_k)$ in this case. On the other hand, if $g_{k'} \leq g/2$ then $f'(g_{k'})$ is in the part of the graph in which the slope is monotonically increasing. In any such function, $f'(g_{k-1} + g_k) > f'(g_{k-1}) + f'(g_k)$. Therefore the function is maximized when $k = 2$. ■

Theorem 8.1 *The protocol of Section 8.2 has a cushion of* $\lfloor n \log n/2 \rfloor q$.

Proof: The final balance is increased only when groups split. To calculate the maximum final balance, we need to aggregate the final balance increase for the group splits in all possible regrouping scenarios that satisfy the constraints of the cushion. These are runs which begin with a group of size n and only allow groups to split.

To calculate the amount lost in a split, consider a bancomat b which regroups from a group g to a group g'. To be conservative, b must assume that all of the members of g not in $SS(g, g')$ have sent a withdrawal which b did not receive. The final balance is increased most when these assumptions are all false. Therefore, if the true final balance of g is B^g, then b calculates the final balance of g as $B^g_b = B^g - (|g| - |SS(g, g')|)q$. The share of this balance that is brought into g' by the members of $SS(g, g')$ is $|SS(g, g')|/|g| \cdot B^g_b$. A similar equation can be derived for each of the groups that result from the split of g. Therefore, when group g splits into k groups $g_1 \dots g_k$, the total balances of the resulting groups at that point is $\sum_{i=1}^{k} |g_i|/|g|(B^g - (|g| - |g_i|)q) = B^g - \sum_{i=1}^{k} |g_i|/|g|(|g| - |g_i|)q$. So, the amount added to the final balance by g splitting is $loss(g) = (|g| - \sum_{i=1}^{k} |g_i|^2/|g|)q$.

Since each group can successively split into smaller groups until there are only groups of size 1, we must add to the above equation the loss from each g_i that results from g splitting. We therefore get the recurrence relation $loss(g) = |g| - \sum_{i=1}^{k} |g_i|^2/|g|)q + \sum_{i=1}^{k} loss(g_i)$. In a run that begins with a group g of size n, the final balance will be the $loss(g)$.

The cushion is defined as the maximum final balance for the runs that are considered. Thus, we need to calculate the maximum for the function $loss(g)$ where $|g| = n$. This maximum must be calculated over all possible splitting scenarios. In Lemma 8.1, we show that the maximum value for this function is $n \log n/2$. Therefore, the cushion for this protocol is $\lfloor n \log n/2 \rfloor q$. ■

8.3 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 notification; 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 notifications 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 notification, it first computes a conservative initial group balance for a hypothetical group in which the withdrawal notification 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 notification 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 $(n^2 - 1)q$. Consider the run in which all bancomats remain connected and all withdrawal notification are sent to i. It will continue to allow withdrawals through $B = qn^2$. Once the final quantum is taken, i will allow no more withdrawals giving a final balance of $(n^2 - 1)q$.

This is a very conservative protocol, and it is an open question whether there is a less conservative version. We discuss this further in the conclusions.

9 Fourth Approach: Shared State, Coordinated Actions

The fourth 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 5.

The protocol is similar to the one of Section 8. 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 requesting 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 8. 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'.

9.1 Formal Description

The event handlers for this protocol are shown in Figure 5. In this protocol, the toggle `SentQuantum` is used to check if there are unstable withdrawal requests. So, in Figure 5, if the bancomat needs

```
  // Local group and balance for this bancomat
 Set group = initial group;
 int balance = initial balance;

// keep track of requests from clients, and whether have outstanding messages
 bool sentQuantum = false;
 int requestedWithdrawals = 0;
----------------------------------------------------------------------------
ClientWithdraw(int k)
    requestedWithdrawals = requestedWithdrawals + k * q;
    TryToSendQuantum();

Regroup(Set newGroup, Set survivors)
    balance = balance  * (|survivors| / |group|);
    group = newGroup;

    sentQuantum = false;
    TryToSendQuantum();

TryToSendQuantum()
    int actualWith = min(requestedWithdrawals,q);
    if (¬sentQuantum && (actualWith > 0))
        sentQuantum = true;
        send(⟨Withdrawal of actualWith⟩);

Receive(⟨Withdrawal of w⟩ from b′)
    if (b′ ≠ b)
        balance = balance - w;

Stable(⟨Withdrawal of w⟩ from b)
    balance = balance - w;
    output(w);
    requestedWithdraws = requestedWithdraws - w;
    sentQuantum = false;
    TryToSendQuantum();
```

Figure 5: *Bancomat* Protocol with Shared State and Coordinated Actions

to send a withdrawal request and it has no such outstanding requests, then it sends a new one.

9.2 Cushion

In Theorem 9.1 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 8.

Theorem 9.1 *The protocol of Section 9.2 has a cushion of* $\lfloor n \log n/2 \rfloor q$.

Proof: The proof for this protocol is similar to Theorem 8.1. We show here that the loss per group is the same as for that protocol.

Consider a group g that splits into k groups $g_1 \ldots g_k$. Each member of g has made a withdrawal request, but none of the withdrawal requests have been notified as stable at any of the bancomats. If bancomat b is in a group that does not include b', then the withdrawal sent by b' in g will be removed from the final group balance of g by b, even though that money is not released by b'. For group g_i, there will be $|g| - |g_i|$ such withdrawals. Therefore, if the true final balance of g is B^g, then b calculates the final balance of g as $B^g_b = B^g - (|g| - |SS(g.g')|)q$. The share of this balance that is brought into g' by the members of $SS(g.g')$ is $|SS(g,g')|/|g| \cdot B^g_b$.

This is the same as the value for Theorem 8.1, and so the same cushion results. ■

9.3 Group Membership Requirements

This solution requires the group membership service to provide total ordering of messages and stability notification, in addition to the property that all of the members of $SS(g,g')$ have delivered the same set of messages while in group g. 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 [13] is sufficient for this solution. Total ordering comes at a cost, however, especially in a wide-area network.

10 Discussion

In this paper, we examined a partition-aware problem, discussed four different approaches to solving partition-aware problems, and compared how well different group membership protocols support solutions to this problem. In this section, we make some observations and raise some questions about these issues.

10.1 Partition-Aware Problems

The definition of partition-aware from [2] that we use in this paper is weak enough to encompass a large set of problems. We were surprised, however, at how hard it was to find a partition-aware problem that was interesting in terms of being sensitive to different partitionable group membership protocols. 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 an appropriate metric, like the cushion, that captures the impact of uncertainty in the global state with respect to partitionable group membership protocols. Indeed at least one of these problems requires *no* communication, and therefore does not require any of the communication properties discussed in 5.

One open question is what other partition-aware problems exist that require or benefit from the different properties provided by the group membership protocols. If there are a large number of such problems, then there may be interesting classes of problems defined by what they require from the partitionable group membership protocols. In this case, it would be worthwhile to identify such classes to aid the choice of which partitionable group membership protocol is best for a problem. If, to the contrary, there are only a few such problems, then it might be worthwhile to design partitionable group membership protocols with these specific problems in mind.

10.2 Different Approaches for Partition-Aware Problems

We examined four different approaches to solving partition-aware problems: one which is not a concurrent solution; one in which processes act autonomously and communicate as infrequently and with as few processes as possible, one that generalizes the state machine approach [23] to partitionable systems, and one that is an intermediate approach; processes act autonomously but broadcast their actions to their connected component.

The first approach was not a concurrent solution. This approach shows how the relative uncertainty in the system can be constrained, and the inherent cost in such a solution. In particular, this solution suffers from a performance bottleneck due to the serialization of the authorization messages.

The second approach is appealing because it uses very little from the group membership service. We were surprised that what little it assumes about message delivery in groups was sufficient to approximately halve the cushion. The properties it requires 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. It allows a bancomat to emit one quantum quickly but subsequent quanta are emitted at the rate that messages sent to the group become application stable. We have not compared the overhead of waiting for messages to become application stable with the overhead of providing total message delivery order, and application stability is an unusual property for group communication systems.

10.3 Group Membership Protocols and the *Bancomat* Problem

The differences between the different group membership protocols were most important for the intermediate approach of Section 8. 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. It would be interesting to try 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* [11]. 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 $O(n \log n)$ 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.

10.4 Other Metrics for Partition-Aware Problems

The cushion metric is not the only metric that one can use to compare the uncertainty that is acquired during the execution of a partition-aware problem. For example, in the *Bancomat* problem one could measure the the amount of money lost due to any group split as compared to the amount lost over an entire run. Such a metric should be meaningful even in runs in which joins are allowed and in which false suspicions can occur. Or, a different approach could be used. For example, it has been suggested to us that one might compare different solutions by using competitive analysis [6] or adaptability [4].

Acknowledgements We would like to thank Özalp Babaoğlu, Karan Bhatia, Phil Gibbons, Idit Keidar, Alberto Montressor and Aleta Ricciardi for their helpful comments on this paper and the research behind it and Russell Impagliazzo for discussions concerning the material in Section 7. We would also like to acknowledge the long discussions we had with Yaron Minsky and Fred B. Schneider and the resulting insight we gained. Finally, we gratefully acknowledge the detailed comments by the anonymous referees.

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. In *Proceedings of the 18th International Conference on Distributed Computing Systems (ICDCS '98)*, Amsterdam, The Netherlands, May 1998. pages 184–191.

[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] P. Berman and J. Garay. Adaptability and the Usefulness of Hints. In *Proc. 6th Annual European Symp. on Algorithms–ESA '98*, vol. 1461, August 1998. pages 271-282.

[5] K. Birman and B. Glade. Consistent failure reporting in reliable communication systems. Technical Report 93-1349, Department of Computer Science, Cornell University, May 1993.

[6] A. Borodin and R. El-Yaniv. Online Computation and Competitive Analysis. Cambridge University Press,1998.

[7] T. D. Chandra, V. Hadzilacos, S.Toueg and B. Charron-Bost. On the Impossibility of Group Membership. In *Proceedings of 15th Annual ACM Symposium on Principles of Distributed Computing*, Philadelphia, 23–26 May 1996. pages 178–187.

[8] F. Cristian. Personal communication.

[9] 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.

[10] S. B. Davidson. Optimism and consistency in partitioned distributed database systems. *ACM Transactions on Database Systems* 9(3):456–481, September 1984.

[11] D. Dolev. Personal communication.

[12] 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.

[13] 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.

[14] C. Fetzer and F. Cristian. Derivation of fail-aware membership service specifications. Technical Report CS96-502, UCSD, November 1996.

[15] 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.

[16] 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.

[17] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. Chapter 5, *Distributed Systems*, Second Edition (S. Mullender, editor), Addison-Welsey 1993.

[18] 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.

[19] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, July, 1978, pages 558-565.

[20] 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.

[21] A. Ricciardi. Using process groups to implement failure detection in asynchronous environments. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing*, Montreal, P.Q., 19–21 August 1991, pages 341–353.

[22] A. Schiper. "Primary partition virtually synchronous communication" harder than consensus. In *Distributed Algorithms (WDAG 8), Lecture Notes in Computer Science 857*, October 1994, pages 39–52.

[23] 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.

[24] J. Sussman. *Group Communication Services versus Wide-Area Networks.* PhD thesis, University of California, San Diego, Department of Computer Science and Engineering, September 1999.

[25] 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.