NASCENT: Network Layer Service for Vicinity Ad-hoc Groups *

Jun Luo Jean-Pierre Hubaux
School of Computer and Communication Sciences
EPFL (Swiss Federal Institute of Technology at Lausanne), CH-1015 Lausanne, Switzerland
{jun.luo, jean-pierre.hubaux}@epfl.ch

ABSTRACT

Many envisioned applications of ad hoc networks involve only small scale networks that we term *Vicinity Ad-hoc Group*s (VAGs). Distributed coordination services, instead of pairwise communications, are primary requirements of VAGs. Existing designs for distributed services in ad hoc networks apply either a layered structure or a vertical integration. While the former approach is efficient for protocol development, the latter improves runtime efficiency. In this paper, we argue that, since distributed services require group-oriented communications, *both* design and runtime efficiency of protocols can be achieved in VAGs, provided that a dedicated network layer service is designed in place of unicast routing protocols. Based on these arguments, we propose NASCENT as a general network layer service for VAGs. NASCENT provides a lightweight VAG membership management along with a routing structure for message passing, and supports concurrent execution of various distributed algorithms. NASCENT is also tailored to cope with the transiency of VAGs. We demonstrate how smoothly distributed algorithms can be built on top of NASCENT. With a complexity-based analysis, we also show that NASCENT greatly improves the runtime efficiency of these distributed algorithms. Finally, through simulations with *ns*-2, we confirm the ability of NASCENT to support the envisioned VAG applications.

Keywords

Ad Hoc Networks, Vicinity Ad-hoc Groups, Network Layer Services, Distributed Algorithms

1. INTRODUCTION

The highly dynamic topology of ad hoc networks suggests that distributed services are often required, because centralized services relying on individual nodes are not dependable enough. In particular, certain applications of ad hoc networks (e.g., wireless multi-player gaming and teamed robots) require primarily distributed services to coordinate the collective actions of nodes in small scale networks, whereas pairwise connections that support stream traffic are barely used. We term the networks implied by such applications *Vicinity Ad-hoc Groups* (VAGs), in order to emphasize their small network scale and group-oriented communication paradigms (i.e., broadcast in nature). Due to these peculiarities, protocol reengineering appears to be necessary.

Currently, protocols for supporting distributed services in ad hoc networks are usually built upon routing protocols (e.g., AODV [31] and DSR [16]). Examples include overlay multicast (e.g., [15, 3, 24]), membership service (e.g., [37]), and resource discovery and management (e.g., [11, 21]). This layered approach, inherited from the Internet architecture, aims at reducing the protocol design complexity; as unicast routing provides the upper layer with a fully connected graph, virtually any distributed algorithm can be built on top. However, this approach can be inefficient for those applications where a fully connected graph is an overkill. In the Internet, this inefficiency is tolerable thanks to unlimited bandwidth, but the problem has to be tackled in ad hoc networks since both bandwidth and power are scarce. An alternative approach [26, 41, 19, 1] suggests that so called *mobility aware* distributed algorithms be built directly upon the MAC layer. Although protocols designed with this vertical integration approach are efficient at runtime, the design complexity is increased because, without a layered structure, low-layer protocols have to be developed separately for individual services.

There is of course a contradiction between these two approaches, but we believe that a trade-off can be found to achieve the *best of both worlds* in VAGs. Since communications in VAGs are group-oriented, it is possible to design a network layer service that includes common enabling mechanisms (e.g., broadcast message routing and membership management), and to leave distinct requirements fulfilled by upper layer distributed algorithms. The protocol design complexity of this approach is similar to that of the layered approach, but the run time efficiency of resulting protocols can be improved to the level of the vertical integration approach. Also, building such a service at the network layer makes it more adaptive to impacts of different mobility patterns and MAC or even physical layer protocols, compared with services based on unicast routing. The small network (or group) size, as well as transient link and node failures of a VAG, can also be exploited. This reduces global membership tracking to local failure detections (similar to [40]), and again simplifies the protocol design and improves the runtime efficiency.

*The work presented in this paper was supported (in part) by the National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS), a center supported by the Swiss National Science Foundation under grant number 5005-67322. (http://www.terminodes.org)

In this paper, we investigate the problem of supporting distributed services in VAGs, bearing specific applications in mind. Our NASCENT (Network lAyer ServiCE for viciNiTy ad-hoc groups) aims at replacing routing protocols to support applications that involve mainly group-oriented communications. NASCENT integrates a *directed acyclic graph* (DAG) with *token circulation*: while the DAG handles message routing and group membership, the circulated token grants temporary control of a VAG to its holders. As a result, NASCENT provides support for various distributed algorithms. Our contributions are: (i) a general network layer service to support various distributed services, (ii) embedded membership service requiring only localized operations, (iii) algorithms to initialize a token-oriented DAG among an emerging VAG, and (iv) algorithms to recompose the DAG upon the VAGs' merging. We present several distributed algorithms that can be easily developed by using the services of NASCENT. In addition, we apply a complexity-based analysis to show the reduced runtime complexity of these algorithms. Finally, we perform simulations with *ns*-2; the results prove that NASCENT is capable of supporting the targeted applications.

The rest of this paper is organized as follows: Section 2 explains our motivation for this work. Section 3 states the problem and the network model. Section 4 presents our NASCENT service. Section 5 gives examples of distributed algorithms built upon NASCENT and proves the efficiency gains in both protocol design and runtime over existing approaches. Simulation results are provided in Section 6. Section 7 surveys related work. Finally, Section 8 concludes the paper.

2. MOTIVATIONS

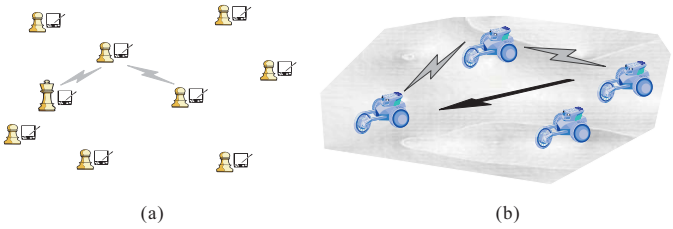
In this section, we first list several envisioned applications. Then we identify the common requirements of these applications, as well as their distinct features. The outcome of the case studies then leads to the motivations of our protocol design.

2.1 Potential Applications

We envision three applications that have immediate needs for both ad hoc networking and distributed coordination services. Note that we only list a few here for brevity. There could be more applications in the same vein.

a. **Wireless multi-player gaming**: Although devices supporting wireless gaming such as the Nokia N-Gage™ game deck do exist, the players either have to pay bills (with GPRS) or are limited in number or by mutual distances (with Bluetooth). Devices based on IEEE 802.11-like wireless MAC gain more freedom, but distributed coordinations become necessary due to the absence of a centralized control. Fig. 1(a) shows a gaming scenario where players are contending for a special position (the *queen*).

b. **Cooperative robotics** [6]: Using teamed robots for unmanned explorations and rescue operations becomes an increasingly tempting application. Therefore, several ongoing researches (e.g. [34, 17]) are focusing on coordinating robots with wireless communication networks. Fig. 1(b) shows an example of exploration robots making an agreement on the moving direction.



(a) (b)

Figure 1: Application examples: (a) wireless multi-player gaming, and (b) exploration robots.

c. **Cooperative driving system** [35]: Vehicles running through critical points, such as highway entrances and blind crossings (crossings without light control), have to be scheduled to share the resources (i.e., those critical points), in order to avoid collisions. Distributed algorithms relying on inter-vehicle communications could be a conflict resolution method performed by the vehicles themselves.

The aforementioned applications share several characteristics. First, the scale of the network is usually small (either geographically or logically or both). This is easy to see because vehicles have no interests in contending a critical resource that is far away from them, and people in one city do not play a wireless game with people in another city (in which case they can play games over the Internet). Secondly, these networks tend to be transient. For example, players join and leave a wireless game spontaneously; and the task dedicated groups of robots might change when finishing a job and starting another. The network topology also changes with time due to node mobility. Thirdly, broadcast is the dominating communication paradigm within the network, which is a consequence of the group-wide coordination explained in the next paragraph. We term such a small network VAG (i.e., *Vicinity Ad-hoc Group*) to depict these peculiarities.

Members belonging to a given VAG have the same goals (performing specific tasks or sharing certain resources), while the group is inherently distributed with each member having only local information. This situation calls for various distributed algorithms to coordinate the behavior of a VAG. In cooperative driving and wireless gaming scenarios, resource allocation and consensus can be useful. Leader election and reliable broadcast (with certain ordering properties) are also needed in gaming. Depending on a given task, a robot team may require one or all of these functions. Moreover, a membership service is generally considered as an important building block of distributed algorithms. However, a service that tracks membership globally and ensures a consistent view at each member does not make sense in transient groups since it could

hardly converge. Replacing global membership tracking with local failure detections, as suggested in [40], results in a light-weight membership service that still guarantees the correctness of algorithms based on the service. Finally, the group-wide coordinations in a VAG require a routing structure that directly supports broadcast for improved efficiency, because the states of individual members have to be conveyed to the whole group.

While the common features suggest a uniform framework, constraints of each application have to be taken into account in order to design functional protocols. On one hand, the mobility pattern of a VAG greatly depends on the targeted applications. Vehicles have high speed but their motions are predictable thanks to the known topology of roads. Game players tend to be relatively static when the game is going on. The movements of robots might be more complex, but the group behavior makes them somewhat foreseeable. On the other hand, the MAC and physical layer protocols could vary according to application requirements and technology developments. Inter-vehicle communication systems may need devices that support a large transmission range and flexible assignment of radio resource [22]. The physical layer of a robot depends greatly on the function of the robot.

2.2 Summary

Following the aforesaid investigations, we summarize our design motivations as follows:

- A dedicated network layer service could be more efficient than unicast routing protocols (which support point-to-point stream traffic) in supporting various distributed algorithms, considering the broadcast nature of communications in a VAG.
- This service should also take care of membership management, because distributed algorithms usually have the same need of membership information, in spite of different intentions.
- The transiency of VAGs requires the service to have the ability of prompt initialization and to cope with frequent topology and membership changes. It also suggests that a global membership tracking would be highly inefficient.
- The service should also be capable of adapting to the change of underlying protocols (e.g., MAC) and to different mobility patterns.

3. GOALS AND MODELS

All the aforementioned facts have motivated us to come up with new networking protocols. To this end, we formally define the problem we want to solve and the considered environments.

We consider small scale ad hoc networks (or VAGs), where group communication is the dominating communication paradigm. Our goal is to develop a group-oriented network layer service, other than a unicast routing protocol, to support distributed services (in particular distributed algorithms such as reliable broadcast and resource allocation) in VAGs. The service should provide consistent interfaces to the upper layer distributed algorithms and make the underlying topology changes (including the changes of membership) transparent to them.

We assume that every node in a VAG has a unique address *addr*. Nodes may fail only by crashing, i.e., stop functioning. Failures are not permanent and can be recovered from.[1] All communications between nodes are assumed to rely on the underlying MAC protocol, so that only nodes within the transmission range of each other can communicate directly and are thus termed *neighbor*s. Each communication link is assumed to be bidirectional and FIFO. A link is reliable for unicast (denoted by a SEND() primitive), but it is unreliable for broadcast (denoted by a BCAST() primitive). The network is modeled as an undirected graph $\mathcal{G} = (V, E)$, with nodes as vertices and an edge existing only between neighbors. The graph changes dynamically and is not necessarily connected. We further assume that the members of a VAG follow group-based movements (e.g., [42]); each member is aware of its mobility group defined by its VAG membership (which is not the case for [42]). This implies that partitions of a VAG seldom occur and that most partitions are transient.

4. NASCENT: GROUP-ORIENTED NETWORK LAYER SERVICE

We present NASCENT in this section. We first overview the architecture of NASCENT, then we briefly justify our design considerations. The protocol is detailed afterward.

4.1 Overview of NASCENT

As shown in Fig. 2(a), NASCENT is a network layer service to replace routing protocols in the protocol stack for VAG members.[2] It acts as a building block of the group-oriented communications in a VAG, and provides support to distributed algorithms such as *Mutual Exclusion* (MX, a special case of resource allocation), *Leader Election* (LE), *Reliable Broadcast* (RB), etc. The protocol architecture contrasts clearly with the layered structure based on unicast routing in Fig 2(b) and the vertical integration approach in Fig 2(c) (introduced in Section 1).

The protocols that compose NASCENT are also shown in Fig. 2(a). The *Local Membership Tracking Protocol* (LMTP) keeps track of the members within 2 hops by exchanging lists of neighbors with neighbors. The *DAG Maintenance Protocol* (DMP) transforms the undirected graph $\mathcal{G}$ into a token-oriented DAG and maintains the DAG when $\mathcal{G}$ is undergoing any changes or the DAG has to be reformed in response to operations in other protocols. The combination of LMTP and DMP acts as the membership service of NASCENT. The *Token Passing Proto-*

[1]This failure model also captures the case where nodes are intentionally switched off.

[2]Members of a VAG can also install other protocols (e.g., unicast routing) at the network layer, but we do not show them in Fig. 2 because the applications supported by these protocols are not the goal of the VAG.

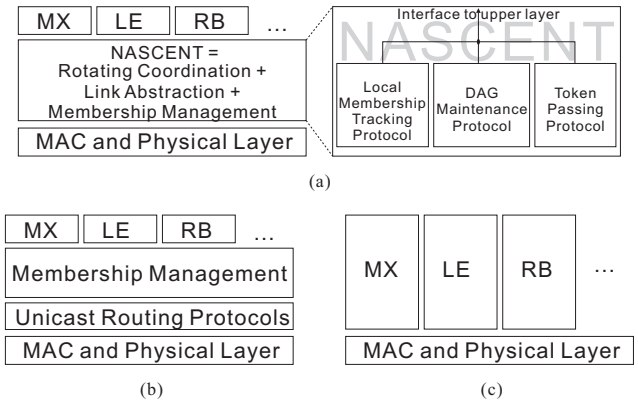


Figure 2: Comparisons between the approach of NASCENT (a) and two designs of existing distributed algorithms (b)&(c).

col (TPP) causes a token to visit every member periodically. The token holder becomes a centralized control of its VAG temporarily, which is a key to building the upper layer distributed algorithms. In addition, each protocol is responsible for the accommodation of certain underlying protocols and mobility patterns. For example, if the MAC protocol already has some mechanisms for neighbor discovery, LMTP can leverage on this and thus reduce its load. Also, specific mobility patterns, such as how vehicles behave at a highway entrance, can be exploited to predict a link breakage.

4.2 Design Rationale

We address the main decisions about our design in this section.

Why local failure detection instead of global membership tracking?

As membership management in wired networks usually involves a global agreement on a certain view of group members, distributed algorithms proposed for ad hoc networks tend to inherit this feature [33, 18] and probe the membership constantly. However, this approach is highly inefficient in a VAG because the network topology and membership change frequently and because possible node failures are often transient. Following the spirit of [40], the membership service of NASCENT ensures that a transient failure does not incur a global membership change.

Why apply a DAG?

A logical structure provides paths to route messages. It also binds all local views together and thus results in a functional membership service. The transiency of VAGs requires the structure to be capable of prompt initialization and to sustain frequent topology and membership changes. We believe that a DAG can meet this need, whereas other structures such as a ring and a tree do not provide enough redundancy to cope with the transiency of VAGs.

Why circulate a token?

The token passing, sometimes called a rotating coordinator paradigm, has long been regarded as an efficient way to support (ordered) reliable broadcast [2, 43, 27] and mutual exclusion [36, 29] in wired networks, then it has also been used in ad hoc networks to implement distributed algorithms [41, 7]. We further observe that a circulated token can be a basis of virtually any distributed algorithm.

Finally, mobility prediction is shown (by, e.g., [42, 20, 37]) to be very important in ensuring the stability of a group, which further guarantees successful terminations of certain distributed algorithms.[3] Considering also the diversity of underlying protocols, the flexibility of accommodating both mobility patterns and underlying protocols will be incorporated into NASCENT, but we do not go into such implementation details in this paper.

4.3 Protocol Details

We describe NASCENT in detail in this section. Each subsection is devoted to a protocol.

4.3.1 Local Membership Tracking Protocol (LMTP)

Each member of a VAG periodically broadcasts a *beacon message* (*bmsg*) including the *id*s of its neighbors. A member, upon receiving the message, infuses the *id*s into its *local view* (*lview*). An *id* is removed from the local view if no *bmsg* from that member is received in τ_b consecutive beacon periods. Note that the *id* of a member is not exactly its address *addr* but a tuple including the address. The format of an *id* is introduced in Section 4.3.2 (where it is used by DMP). Fig. 3 illustrates the exchange of *bmsg*s as well as the format of a *bmsg* and *lview*. Through this information exchange, each member is aware of other members within 2 hops and also synchronizes with its neighbors. Tracking membership only within 2 hops ensures the freshness of the membership. The membership information is valid at least $k-1$ beacon intervals before it is received if the membership is tracked within k hops. The high dynamics of VAGs suggest that a large value of k could not guarantee the correctness of the membership information. Also, tracking membership with a large radius puts a heavy load on the network. Compared with neighborhood tracking, the 2-hop membership tracking gains much more information with only slightly increased overhead, since

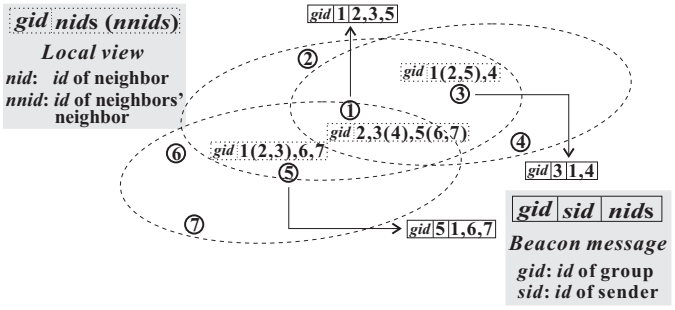


Figure 3: Beacon-based local membership tracking (*addr*s instead of *id*s are shown to save spaces).

[3]The necessity of group stability is often implied by assumptions like "a failure occurs only after the group recovers from the previous failure".

both schemes need beacons but only differ in the size of the beacon message. The GBCast primitive (explained in Section 5.1.2) takes advantage of this information gain to reduce its costs.

Note that LMTP replaces any beacon mechanisms implemented by an underlying MAC protocol (e.g., IEEE 802.11). Moreover, an incoming *bmsg* could be filtered according to certain policies. An example of such policies is *Safe Distance* proposed in [37].

4.3.2 DAG Maintenance Protocol (DMP)

We propose new algorithms to initialize a token-oriented DAG in a newly formed VAG and to recompose the DAG upon the VAGs' merging. To cope with topology changes due to node mobility, we reuse the *partial reversal method*, referred later as PRever primitive, described in [8].

We let the *id* of a member i (i.e., $addr = i$) be a triple $[\alpha_i, \beta_i, i]$. These *id*s, ranked lexicographically ($id_i > id_j \Leftrightarrow \alpha_i > \alpha_j \vee (\alpha_i = \alpha_j \wedge \beta_i > \beta_j) \vee (\alpha_i = \alpha_j \wedge \beta_i = \beta_j \wedge i > j)$), form a total order sequence. Member i considers a communication link with member j as an *outgoing* edge if $id_i > id_j$, otherwise the link is an *incoming* edge. As a result, the communication graph $\mathcal{G}$ is transformed into a DAG.

Initialization

Initially, the *id* of member i is set to $[0, 0, i]$, so the DAG might have more than one sink. In the example of Fig. 5(a), members 1,2, and 3 are all sinks. Since we let a sink hold a token, this situation does not guarantee the uniqueness of the token in a VAG. In order to remove those superfluous sinks (e.g., members 2,3), each member adjusts its *id* according to the incoming *bmsg*s. Algorithms describing these adjustments are shown in Fig. 4. Each member initializes the *gid* in *lview* (refer to Fig. 3 for details) with its *id* (line 2). Upon receiving a *bmsg*, each member checks if there exists a *gid* that is smaller than the one it knows (line 4). The lower *gid* is then taken and the *id* is adjusted properly (lines 5–6). Fig. 5(b) shows the situation after each member broadcasts the first *bmsg* following the initialization of its $lview.gid$. At some point in time, *gid*s of different sinks will arrive at the same members. These members form a *collision region*, shown in Fig. 5(c). Members belonging to a collision region choose the sink with lower *id* as the token holder and confirm the outgoing edge to it; the following *bmsg*s will reverse the edge to other sinks. This procedure continues until all superfluous sinks are removed, which terminates the algorithm and results in a token-oriented DAG (with member 1 being the **initial** token holder), as shown in Fig. 5(d).

```
1: procedure Init(id_i)
2:     lview_i.gid ← id_i; init_i ← true
3: upon Recv(bmsg) do
4:     if (bmsg.gid < lview_i.gid) ∧ init_i then
5:         lview_i.gid ← bmsg.gid
6:         id_i.β ← bmsg.sid.β + 1
```

Figure 4: Initialization at member i

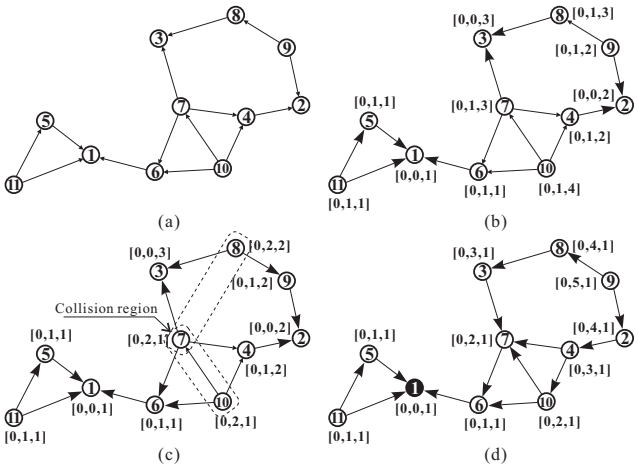


Figure 5: Initialization of a token-oriented DAG. The *gid* and *id* of each member is put in a compact way to save spaces, such that the first two elements are α and β of an *id* and the third element is the *addr* of a *gid*.

We prove the correctness of this algorithm by showing that the following two properties of the algorithm hold.

Property 1. *The algorithm eventually terminates.*

Proof. The $lview.gid$ of each member will be set to the smallest *id* in the network after some time. The algorithm terminates because of the conditional statement in line 4. □

Property 2. *The resulting DAG has only one sink.*

Proof. Assume, in contradiction, that there is an extra sink whose *id* is not the smallest one after the algorithm terminates. Since the algorithm has terminated, the $lview.gid$ of this member is set to the smallest *id* in the network. Considering that the initial $lview.gid$ is larger than the final one, the algorithm must reach line 6 once exactly before the algorithm terminates, which implies that it has an outgoing link. The link will not be reversed after that, because the $lview.gid$ at the other end of this link has already been the smallest *id* (i.e., the algorithm has terminated), a contradiction. □

Although the algorithm terminates correctly, individual members have no knowledge about when the termination happens since the algorithm is executed in a distributed manner. It is important to let every member be aware of the termination in order to make progress in other protocols (e.g., TPP). For this purpose, we force the algorithm to stop (i.e., set the flag *init* to false) after a given time span by using a timer. The timeout value of this timer is set according to the time complexity of the algorithm (refer to Section 5.2 for details). The accuracy of this value is not so relevant because, even if the algorithm is stopped too early due to an improper value, the resulting token collision will be solved by TPP.

Merging

With proper extensions, the algorithm of Fig. 4 lines 3–6 handles also merging. The extended algorithm is shown in Fig. 6. Whenever a node belonging to an established

```
1: upon RECV(bmsg) do
2:    /* Fig. 4 lines 4–6 */
3:    if (bmsg.gid < lview_i.gid) ∧ ¬init_i ∧ ISMERGE() then
4:       init_i ← true; init_timer_i ← 0
```

Figure 6: Merging at member i

VAG receives a *bmsg* whose *gid* is smaller than its own (line 3), it switches back to the initialization phase (line 4). The border members of this VAG that meet nodes from the other VAG form the collision region in Fig. 5(c). The algorithm is guaranteed to terminate and results in only one sink whose *gid* is the smallest. We note that a merging does not always take place when VAGs meet each other. It is the duty of a given application to decide whether to start a merging procedure, by responding to the callback procedure ISMERGE().

Maintenance and Partition Detection

The PREVER primitive that maintains the token-oriented DAG (briefly summarized in Fig. 7) is the partial reversal method in [8]. This primitive is invoked by a VIEWCHG

```
1: upon VIEWCHG do
2:    if ¬init_i ∧ (id_i < nid, ∀nid ∈ lview_i) then
3:       id_i.α ← min_{nid∈lview_i}{nid.α} + 1
4:       if ∃nid ∈ lview_i s.t. nid.α = id_i.α then
5:          id_i.β ← min_{nid∈lview_i ∧ nid.α=id_i.α}{nid.β} − 1

6: upon RECV(token) from id_j do
7:    id_i.α ← id_j.α; id_i.β ← id_j.β − 1

8: upon SEND(token) to id_j do
9:    id_j.α ← id_i.α; id_j.β ← id_i.β − 1
```

Figure 7: PREVER at member i

event resulting from any changes of the *lview* (lines 1–5) or a token transfer (lines 6–9). Note that the response to a token transfer is not defined in [8], but our extension follows the general guidelines discussed in [8] and thus guarantees that only a unique sink holds the token. It is known that the algorithm becomes unstable in partitions that disconnect from the token holder. Partition detection mechanisms (e.g., TORA [30]) can be one solution. We will describe an alternative solution based on a timer in Section 4.3.3.

Join and Leave

A node intending to join a VAG puts a *join indication* in its *bmsg*s. If the *id* of a joining member does not make it to be a sink, then nothing happens except that the member infuses its *lview* with the members nearby and sets its *lview.gid* to the *gid* acquired from these neighbors. Otherwise the PREVER primitive is called to reverse some of its links. Members broadcast *leave indication*s upon leaving. In particular, a token holder should hand on its token before leaving. The PREVER primitive for the remaining members automatically recovers the token-oriented DAG.

Functionalities of LMTP–DMP Combination

The combination of LMTP and DMP acts as both a routing structure and a membership service. As a routing structure, this combination facilitates the routing of broadcast messages from a token holder and unicast messages towards it. As a membership service, this combination limits the impact of any membership changes to only a scale of 2 hops with LMTP, while bridging the gaps between individual views with DMP. This scheme makes sense in the cases of both transient failures[4] and permanent membership changes (e.g., nodes join and leave). The reason is that certain distributed algorithms (e.g., resource allocation) do not necessarily need a global membership view as long as all VAG members are informed of the outcomes of algorithms executions. Whereas membership changes can potentially be propagated for applications that require the information. Detailed usages of these functions can be found in Section 5.1.

4.3.3 Token Passing Protocol (TPP)

TPP circulates a *token* within a VAG. The token is a message that contains certain system states. A token holder acquires and modifies these states, and thus acts as a temporary coordinator, which facilitates various distributed coordinations. We propose two algorithms, TPP-Q and TPP-R, for circulating the token. In TPP-Q, the token circulation is performed by a distributed queueing system. This system guarantees that the token repetitively visits each member with a period linear with n. TPP-R applies a more heuristic method based only on local recency information, which brings less overhead but comes with a larger worst-case upper bound of the circulation period. We provide only intuitive ideas of TPP in this section and leave detailed descriptions to the Appendix.

Token Circulation with a Queueing System (TPP-Q)

As a follow-up of Fig. 5, Fig. 8 provides an example of one token circulation period with TPP-Q. Each VAG member is equipped with a queue. Upon finishing initialization, each member, except the token holder, enqueues its own request of the token and also forwards a request to a member with minimum *id* in its local view. A request contains the requester's address *saddr* and a count *epoch* that denotes which cycle of the request is for. Fig. 8(a) shows that the distributed queueing system actually builds a rooted spanning tree. Then, as exhibited by Fig. 8(b),(c),(d), the token will travel along the tree edges. They also show that the *epoch* of every queued request is stepped up by one after the token has visited all members. As a consequence, the token will repeat the same trajectory that it follows in the first cycle and thus visit each member periodically. We distinguish the case where a member is **visited** by the token from the case where a member **receives** the token. A member is considered to be visited only if its request is the first in the queue when it receives the token.

The spanning tree has to be repaired when some tree edges are broken due to topological changes. Since the under-

[4]This type of failures includes node failures (e.g., operating system crashes), link failures (due to, e.g., interference), and node mobility.

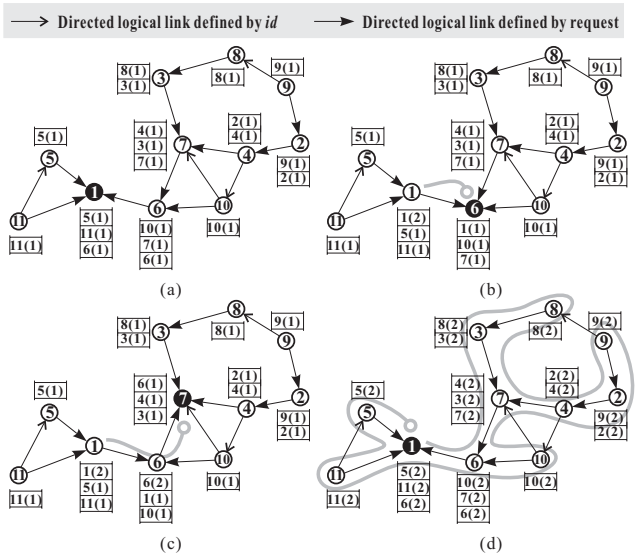


Figure 8: One cycle of token circulation by TPP-Q. Each request in the queue is represented by $saddr(epoch)$**, and the queue is FIFO with its header at the bottom.**

lying token-oriented DAG is repaired by DMP, a member with a broken outgoing edge in the tree simply replaces the edge with another outgoing edge in the DAG. Detailed operations are presented in the Appendix.

Token Circulation with Recency Information (TPP-R)
If TPP-R is used for token circulation, each VAG member keeps an array that records the time when the token visited each neighbor recently. A member that **receives** the token is **visited** by it only if the member has the least recency record among all neighbors, otherwise it forwards the token to a neighbor with the least recency record. The recency information is piggybacked with the token to inform a member about the records of its neighbors. Although TPP-R usually has a smaller token circulation period than that of TPP-Q (because shortcuts can be exploited to avoid backtrackings), the worst-case upper bound of this period is much larger (refer to Section 6.4 for an example).

For both cases of TPP-Q and TPP-R, a timer is set after a token holder releases the token, in order to detect group partitions. The timer times out if the member does not receive the token within a given time span. This leads to a new initialization phase. If partitions really happen, all members that are disconnected from the token holder will enter the initialization phase and thus regenerate a token for each connected partition. If it is a false positive, the member can either rejoin the previous VAG or merge with the VAG, depending on whether an application allows merging or not (see Fig. 6 line 3). Since false positives make a VAG unstable, it is crucial to set the timeout value in order to reduce the probability of false positives. The following property of TPP-Q facilitates this value setting.

PROPERTY 3. *The time for a token to visit all VAG members in one cycle is upper bounded by* $\hat{T} = 2D_{max}nT_t + nT_s$ *if the VAG remains static during that time period. Here* D_{max} *is the degree of the undirected graph* $\mathcal{G}$*,* n *is the cardinality of the VAG,* T_t *and* T_s *are the one way* ***t****ransmission time and the time for the token to* ***s****ojourn at a visited member, respectively.*

PROOF. The distributed queue maintained by TPP-Q creates a rooted spanning tree $\mathcal{T} = (V, E_{\mathcal{T}})$. The exact time to travel all vertices in $\mathcal{T}$ is $2|E_{\mathcal{T}}|T_t + |V|T_s$, where $|E_{\mathcal{T}}| < D_{max}n$ and $|V| = n$. Note that the upper bound is tight enough if $T_t \ll T_s$. □

Although TPP-R has a larger worst-case upper bound, $\hat{T}$ works well in most cases. Therefore, we set the timeout value to $\hat{T}$, and the parameters can either be adaptively changed on-line or be estimated off-line (e.g., the maximum number of roles in a game is known to every player). The way to make on-line adaptations or off-line estimations depends on specific applications, so NASCENT provides the upper layer with an interface to set $\hat{T}$.

5. DISTRIBUTED ALGORITHMS ON TOP OF NASCENT

In this section, we sketch the implementation of several distributed algorithms on top of NASCENT. We also perform a complexity-based analysis on NASCENT and these algorithms. Our goal is to demonstrate how NASCENT simplifies the design of distributed algorithms and improves their runtime efficiency. We avoid detailed descriptions and correctness proofs for brevity.

5.1 Algorithm Descriptions

We briefly describe several distributed algorithms built upon NASCENT. It is straightforward to see that the design of these algorithms is simplified thanks to the use of services provided by NASCENT.

5.1.1 Resource Allocation

A special case of resource allocation is mutual exclusion that allows only one member to access a unique system resource each time. The authors of [41] demonstrate that this problem can be solved by a token-based algorithm. Since NASCENT also circulates a token, it is very straightforward to build a mutual exclusion algorithm on top of NASCENT. Here we consider a more general resource allocation problem, where several instances of a single system resource exist and only one member can access one instance each time. We also distinguish between *local multi-instance* and *distributed multi-instance* resource allocation. In the former case, all instances of the resource are geographically close to each other. Cooperative driving at a critical point (Section 2) is a good example, where lanes that run parallel are instances of a resource (the critical point). The latter case is more broad in sense, because it applies to many coordination functions in multi-player games and teamed robots.

The local multi-instance resource allocation can be solved by simply extending the mutual exclusion algorithm: a token holder, instead of allowing only itself to access the

resource, also grants permission to its neighbors for accessing other instances of the resource. We hereafter focus on the distributed multi-instance resource allocation. Let

```
1: upon ReqRes() do
2:    req_res_i ← true

3: upon RelRes() do
4:    req_res_i ← false

5: upon Recv(token) do
6:    if req_res_i ∧ (∃i ∈ [1, m] s.t. token.sub_tk[i].id = null) then
7:       token.sub_tk[i].id ← id_i
         /* now access the resource and release the token */
8:    if ¬req_res_i ∧ (∃i ∈ [1, m] s.t. token.sub_tk[i].id = id_i) then
9:       token.sub_tk[i].id ← null
         /* now release the token */
```

Figure 9: Request and release a resource instance at member i

the token carry m *sub_tk* that represent m instances of a certain resource. Fig. 9 shows an algorithm for allocating these instances. A member intending to access the resource indicates its requirement (lines 1–2). The member can access the resource only if it receives the token and at least one instance has not been claimed by other members (lines 6–7). After finishing the resource access, the member signals this and releases the *sub_tk* it claimed upon receiving the token (lines 3–4 and 8–9). In addition, the token can also carry a queue that orders the resource requests to ensure fairness.

5.1.2 (Ordered) Reliable Broadcast

A reliable broadcast ensures that a message broadcast by a member is delivered by all correct members [12]. Our solution built upon NASCENT is similar to what is proposed in [27]. As shown in Fig. 10, a member broadcasts

```
1: procedure RBCast(msg)
2:    SMsgBuffer_i ← SMsgBuffer_i ∪ {msg}

3: upon Recv(token) from id_j do
4:    if SMsgBuffer_i ≠ ∅ then
5:       for all msg ∈ SMsgBuffer_i do
6:          GBCast(msg)
7:          token.view[id_i][msg.mid].recv ← true
8:          SMsgBuffer_i ← SMsgBuffer_i/{msg}
9:    for all msg s.t. token.view[id][msg.mid].recv ≠ true do
10:      if msg ∈ RMsgBuffer then
11:         token.view[id_i][msg.mid].recv ← true
12:      else
13:         request msg from id_j
14:   if ∄nid s.t. token.view[nid][msg.mid].recv = false then
15:      Deliver(msg)      /* to the upper layer */

16: upon Recv(msg) from id_j (sent by BCast) do
17:   if msg ∉ RMsgBuffer_i then
18:      RMsgBuffer_i ← RMsgBuffer_i ∪ {msg}
19:      GBCast(msg)
20:   bcast_view[msg.mid] ← bcast_view[msg.mid] ∪ {id_j}

21: procedure GBCast(msg)
22:   if ∃nid ∈ lview s.t. nid > id_i ∧
      nid ∉ ⋃_k id_k.lview, ∀id_k ∈ bcast_view[msg.mid] then
23:      BCast(msg)            /* see Section 3 */
```

Figure 10: Reliable broadcast at member i

a message only when holding the token (lines 1–8), by invoking the GBCast primitive (lines 21–23). A token holder also sets the *recv* flag corresponding to a received message to true (assume that *mid* uniquely identifies a message). A missed message is requested from the previous token holder. The message is delivered to the upper layer only if all members have received it (lines 9–15). A member stores a broadcast message received for the first time and rebroadcasts it, also by the GBCast; the member also memorizes the sender of each received message (lines 16–20). The GBCast primitive leverages on the token-oriented DAG and on the neighborhood information in *lview* to reduce collisions in the MAC layer and gains reliability and efficiency over flooding. It is also more robust than a tree-based multicast protocol, since the VAG members are linked by a (directed) mesh. The reliability of GBCast is further enhanced considering that only one member (the token holder) can initiate a group-wide message dissemination. As a consequence, negative acknowledgements (line 13) are rarely sent by a member, which results in an efficient reliable broadcast algorithm. If a member wants to broadcast an urgent message without waiting for the token, it can send the message to the current token holder and ask for a proxy-broadcast. This is possible because the DAG supports unicasts from all members to the token holder.

Since messages are ordered in the token, a slightly modified version that requires each member to deliver messages according to their order in the token solves the total order broadcast problem (i.e., all correct members deliver messages in the same order). In addition, FIFO order and causal order [12] are implicitly guaranteed. We note that the variable *view* in the token actually maintains a global view of membership. It means that NASCENT has the potential to support global membership tracking; this service is, however, not embedded in NASCENT because it is not necessary for all distributed algorithms (e.g., resource allocation).

5.1.3 Leader Election

Solutions to the leader election problem become trivial if both mutual exclusion and reliable broadcast are solved. As defined in [26], leader election needs to ensure that a group whose topology remains static sufficiently long will eventually have exactly one leader. The problem is similar to mutual exclusion, because both problems are concerned with a special member. They differ in that leader election requires other members to be informed about the existence of this special member, which is exactly what reliable broadcast does. Our solution is the following: a leader candidate who acquires the token before other candidates declares itself to be the leader; it then broadcasts its *id* with the RBCast primitive (Fig. 10). An alternative way is to put its *id* into the token, thus trading latency for reduced communication costs.

5.2 Complexity Comparisons

In this section, we defend our claim that NASCENT improves the runtime efficiency of the described algorithms. For this purpose, we show that the complexity of maintaining NASCENT is of the same magnitude as unicast routing protocols, and that the complexity of running distributed algorithms upon NASCENT is greatly reduced compared

to the same algorithms based on unicast routing protocols. Note that the former complexity is measured against one network perturbation (i.e., node or link failures) and the latter is evaluated with respect to one specific operation.

We make use of *time complexity* (TC) and *communication complexity* (CC) to quantify the performance of protocols. We refer to [9] for detailed definitions of these two terms and related synchrony assumptions. In Table 1, we compare the performance of several routing protocols (DSDV [32], AODV [31], DSR [16], GB [8], and TORA [30]) with NASCENT. The complexity computations of routing protocols are borrowed from [4, 39]. Each protocol is evaluated in two situations, namely *initialization* and *postfailure*, distinguished by the subscript of performance terms (i and f, respectively). The comparisons show that the post-failure complexity of NASCENT is comparable to those of routing protocols, whereas the initialization complexity is slightly higher. However, directly comparing table driven protocols (DSDV and NASCENT) with on-demand protocols (AODV, DSR, TORA) is unfair, since the complexity of an on-demand protocol is evaluated for each route. In the case of group-oriented communications (where one node needs routing paths to every other node), on-demand protocols would incur much higher complexity than table driven protocols because, in the worst case, an actual CC is the one shown in Table 1 times $n(n-1)/2$. Therefore, DSDV is the only rival of NASCENT in the considered scenarios of this paper.

	DSDV	AODV; DSR	GB; TORA	NASCENT
TC_i	$O(d)$	$O(2d)$	$O(2d)$	$O(2d)$
TC_f	$O(d)$	$O(2d)$	$O(l)$; $O(2d)$	$O(l)$
CC_i	$O(n)$	$O(2n)$	$O(2n)$	$O(2n+n)^*$
CC_f	$O(n)$	$O(2n)$	$O(x)$; $O(2x)$	$O(2x+x)$

n = Number of node in the network
d = Network diameter
x = Number of nodes affected by a topological change
l = Diameter of the affected network segment
* CC of DAG maintenance + CC of token request forwarding
Note that the second part appears only if TPP-Q is used.

Table 1: Performance comparisons between routing protocols and NASCENT.

Note that although the TC_fs of NASCENT and GB are equal because NASCENT applies GB to cope with failures in connected networks, the same TC_is of NASCENT and GB do not suggest the same protocols. Actually, NASCENT transforms a multi-sink DAG into a token-oriented DAG, whereas GB converts a destination-disoriented DAG into a destination-oriented one. The higher CC of NASCENT compared to GB is due to the token circulation, since extra token request messages have to be sent. In addition, Table 1 seems to suggest that TORA has a higher complexity compared to GB in the case of failures, but this is not true because TORA can cope with network partitions with the added complexity (while GB has an infinite complexity in the cases to partition). NASCENT uses a timer to detect network partitions and reinitiates the disconnected components. Therefore, the CC_f of NASCENT is the complexity of initializing a component with x nodes.

According to the comparisons of Table 1, it is hard to prove that NASCENT is better than DSDV for supporting distributed services in VAGs. However, if we look at distributed algorithms that make use of the services provided by these two protocols, the superiority of NASCENT becomes clear. In Table 2, we use reliable broadcast as an example to show the complexity of building distributed algorithms on top of DSDV and NASCENT. The algorithm proposed in [12] (we refer to it as HT_RB to credit the authors) is a representative of non-token-based solutions. It can be roughly described as "each member, upon first receiving a message, broadcasts it to other members". This explains why it has a complexity of $O(n)$ for both TC and CC. However, it is much more expensive than a token-base approach, considering that each message is broadcast. If only the number of messages is counted, the token-based algorithm we proposed in Section 5.1.2 has quite similar CCs when running on top of DSDV and NASCENT, i.e., 1 broadcast message (Fig. 10 line 6) and $3n$ unicast messages (1 token passing message, 1 negative ack and 1 response for each member) in the worst case, while the algorithm has to maintain a DAG with x extra messages if it is based on DSDV. However, a unicast message in the case of DSDV is transmitted through multi-hop routing and DSDV does not provide an efficient way to do broadcast. Therefore, the algorithm based on NASCENT is much more efficient considering the single-hop unicastings and the collision avoidance GBCast (Fig. 10 lines 21–23). Note that although the messages used to pass the token are a part of NASCENT, they are counted in Table 2 since these messages are involved in operations (e.g., broadcasts) instead of coping with network perturbations.

	DSDV+HT_RB	DSDV+RB	NASCENT+RB
TC	$O(n)$	$O(n)$	$O(n)$
CC	$O(n)^*$	$O(3n+x)^\dagger + 1^*$	$O(3n)^\ddagger + 1^*$

HT_RB: the reliable broadcast protocol describe in [12]
RB: the reliable broadcast protocol describe in Section 5.1.2
* Broadcast † Multi-hop unicast ‡ Single-hop unicast

Table 2: Performance comparisons between reliable broadcast protocols built upon DSDV and NASCENT.

Actually, there are three other network layer services that would compete with NASCENT: flooding (including its optimized version such as [23]), ack-based reliable broadcast (e.g., [13]), and multicast (e.g., MAODV [38], ODMRP [20], ADMR [14], and DCMP [5]). The complexity of flooding is comparable to NASCENT+RB in the case of broadcast (but without any reliability guarantee), but 2 flooding messages and n (multi-hop) unicast messages are needed to achieve 1 mutual exclusion [28]. With NASCENT, up to n mutual exclusions can be achieved with n (single-hop) unicast messages (which correspond to the cost of circulating the token). Ack-based reliable broadcast has a large TC in general (because an ack can reach the sender much later than others) and, similar to flooding, it is expensive to build other distributed algorithms based on such protocols. Multicast protocols perform broadcasts within a subset of network nodes (different from VAG sce-

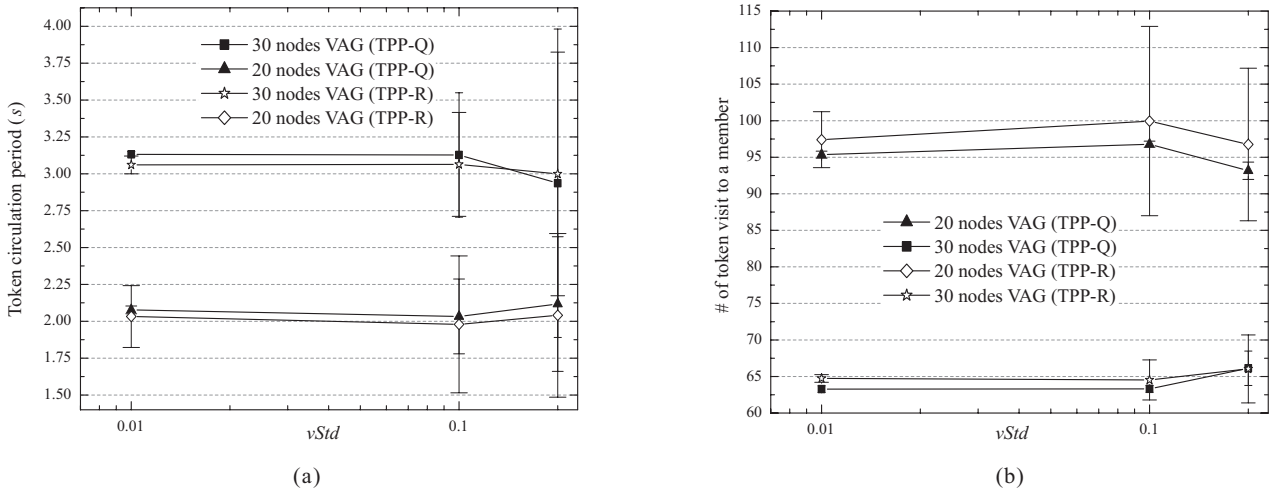


Figure 11: The distributions of (a) token circulation period and (b) number of token visits to a member under different *vStd*s, with T_s = 100ms.

narios) in a more light-weight manner, but it is hard to ensure reliability with these protocols. In addition, some protocols (e.g., MAODV and ADMR) are based on tree structure; it is more costly to maintain a tree than a DAG in networks of dynamic topology. For example, the CC_f in connected networks is $O(n + x)$ for MAODV to maintain a tree and $O(x)$ for NASCENT to maintain a DAG.

6. SIMULATIONS

Having provided complexity-based comparisons to justify the benefit of NASCENT, we use simulation to verify the ability of NASCENT to support the envisioned applications.

6.1 Simulation Setup

We take *ns-2.26* as the simulation platform. We use IEEE 802.11 with 2Mbps transmission rate as the wireless MAC but reduce the nominal range from 250m to 100m, in order to make our simulations more realistic. We adopt the two-ray ground reflection model as the radio propagation model.

We simulate VAGs with 20 and 30 nodes in a square area of 1km^2, during 200 seconds of simulated time. Initially, members of a VAG are randomly distributed within a region of 250m×250m such that they are in the "vicinity" of each other. The movement pattern is defined by a group mobility model similar to [42] where the following process is repeated[5]: a VAG chooses a *group speed* uniformly distributed between zero and a maximum value as well as a random direction, then each member chooses a velocity following a 2-D normal distribution parameterized by the group speed and a standard deviation and begins to move for a certain time period. Upon timing out, the VAG remains static for some pause time. Both moving time and pause time are described by a uniform distribution between zero and a maximum value. Note that assuming a random moving time instead of an arbitrary destination partially solves the problem of decreasing average speed pointed out by [44]. In this mobility model, the only parameter that has a significant impact on the performance of NASCENT is the standard deviation of individual member velocities (*vStd* hereafter). Therefore, we fix the maximum value of group speed, moving time, and pause time to 20m/s, 50s, and 10s, respectively, and test NASCENT only under different *vStd*s.

We set the beacon period to 200ms and test NASCENT under T_s = 100ms and 10ms (defined in Section 4.3.3). We also assume a 18-byte beacon message and a 50-byte token message. Members perform NASCENT initialization within the first 2 seconds, then the member with the smallest *id* (i.e., the sink) generates a token and starts to circulate it. Each simulation is carried out 10 times with different scenario files.

6.2 Stability of Token Circulation

In a static network, we say that the token circulation is stable if the token visits every member in a cycle and the period of a cycle remains constant. According to PROPERTY 3, NASCENT meets this requirement. However, no protocol can be qualified with this criterion in networks with a dynamic topology and membership, notably because the token cannot visit a member temporarily broken from the network in the current cycle. Therefore, we take an alternative criterion: a token circulation is stable if (i) the distribution of the circulation period has a small variance and (ii) the token visits each member infinitively often. The stability of token circulation under different *vStd*s is a major performance index of NASCENT, because a stable token circulation guarantees the correctness of upper layer distributed algorithms and indicates an effective membership tracking.

Fig. 11(a) and (b) show the verification of the conditions

[5]This model is not compatible with some of the cooperative driving scenarios described in Section 2. Dedicated traffic modeling should be applied for detailed investigations on these cases, so we leave it as future work.

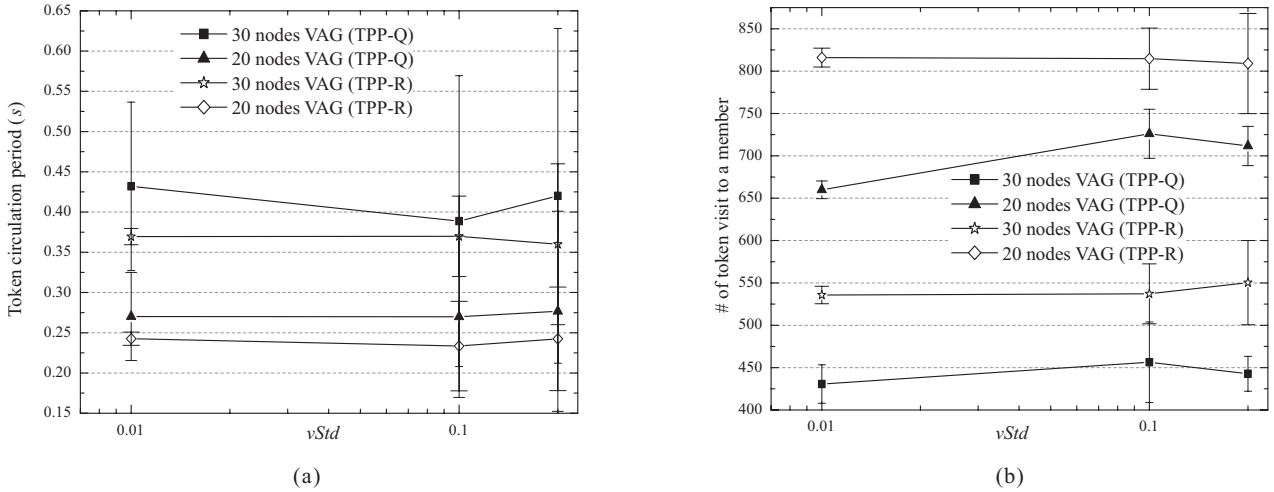


Figure 12: The distributions of (a) token circulation period and (b) number of token visits to a member under different *vStd*s, with T_s = 10ms.

(i) and (ii), respectively. The value of *vStd* in the figure is the ratio between a standard deviation and a mean value (the group speed for the polar distance of the velocity and π for the polar angle). Fig. 11(a) shows distributions of the circulation period, and Fig. 11(b) presents distributions of the number of token visits to a member (a way to check condition (ii) within limited simulation time). Each distribution is characterized by a mean value and a standard deviation. The mean values of the circulation period are all quite close to nT_s, which matches the prediction by PROPERTY 3 ($2D_{max}nT_t$ can be neglected if $T_t \ll T_s$). In the quasi-static scenarios with *vStd* = 0.01 (e.g., a group of wireless game players in a moving train), the token is stably circulated for both 20 and 30 node VAGs, since the standard deviations of the circulation period and the number of token visits to a member are relatively small (mainly due to the fluctuation of wireless links). The stability is modestly degraded in scenarios with *vStd* = 0.1 and 0.2 (real life examples could be teamed robots).

In 20 node scenarios, the mean value of the token visit number increases (along with a decrease of the mean value of the token circulation period) when *vStd* increases from 0.01 to 0.1. This is due to the link breakage between a few peripheral members and their VAGs (note that although NASCENT allows partitioned components to have their own token, individual members broken from a VAG are considered to be leaving). Actually, we have to adjust the transmission range of wireless MAC to 120m for *vStd* = 0.2, in order to keep the network from breaking into many small components (where no application would make sense). This exhibits an imperfection of the mobility model but not of NASCENT, because the movement of a VAG member would be more coordinated than the stochastic process that we use for modeling.

6.3 Circulating Token with Smaller T_s

Fig. 11(a) shows that the token circulation period of NASCENT is in the order of seconds. The main reason is that we allow the token to stay T_s = 100ms at each member in a cycle. Fig. 12 shows that, with T_s = 10ms, NASCENT provides a circulation period of several hundred milliseconds. In practice, applications need a certain amount of time to process backlogged operations upon acquiring the token. Therefore, the value of T_s (and thus the circulation period) is defined according to the processing capacity of given devices.

For wireless multi-player gaming, T_s = 10ms is a reasonable value, because the device needs only to exchange some state information with the token. Therefore, the resulting circulation period is short enough to prevent impatient players from giving up the game. In the cases of cooperative robotics, T_s might need to be relatively long considering that the behaviors of a token holder could involve mechanical movements. Fortunately, a circulation period in the order of seconds is tolerable, again due to the low-speed mechanical movements involved in the applications (an extreme case is exhibited by the Mars Exploration Rover that has a top speed of 5cm/s).

6.4 Comparing TPP-Q with TPP-R

When network topology changes dynamically, both TPP-Q and TPP-R perform well in most cases. However, we have observed in simulations that there exist some special situations where these algorithms fail to guarantee the stability of token circulation. For example, in Fig. 13(a), member a cannot pass the token to member b due to a link breakage, but b is not aware of this until the breakage is detected by LMTP after τ_b beacon periods. TPP-Q, in this case, leads to a virtual partition since members that have sent requests to b will not receive the token in the current cycle although the network is connected. A worst-case scenario for TPP-R is shown in Fig. 13(b). Assume that *addr* is used to break a tie, the token will follow the path $a \rightarrow b \rightarrow c \rightarrow \cdots$. Therefore, the token will not visit members to the right side of a before it revisits all members to the left side of a. The significance of the effect of the virtual partition shown in Fig. 13(a) depends on the values of T_s and the beacon period. If they are compara-

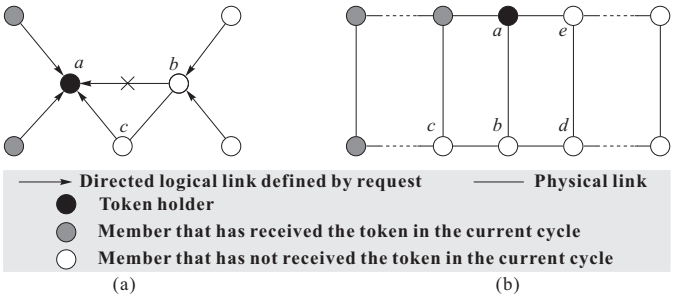


Figure 13: Special cases where (a) TPP-Q and (b) TPP-R fail to guarantee the stability of token circulation.

ble or T_s is larger, the effect becomes less significant and TPP-Q outperforms TPP-R, otherwise TPP-R wins. This conclusion can be observed in Fig. 11 and 12. The conclusion also suggests different application domains (defined by required T_s and beacon period) for TPP-Q and TPP-R.

7. RELATED WORK

Several research areas are related to our proposal, including network layer services (e.g., broadcast/multicast routing) as well as DAG or token-based distributed algorithms. Note also that our proposal has the potential to support *mobile agent* [10] in VAGs (in-depth discussions are omitted). Having compared NASCENT with various network layer services in Section 5.2, we now focus our discussion on relevant distributed algorithms for ad hoc networks.

To our best knowledge, only two distributed algorithms for ad hoc networks [26, 41] apply DAG as their basis. They are the main inspirations of our proposal. Malpani et al. [26] propose a leader election algorithm based on a DAG, and Walter et al. [41] show that a DAG also supports mutual exclusion. In both cases, the sink of the DAG always plays a special role (a leader in [26] and a token hold that gains the access to a resource in [41]). However, the authors do not mention that a DAG could be a common basis for other distributed algorithms. Also, [41] considers static groups without initialization and recomposition of DAGs upon group forming and merging, while [26] applies a mechanism similar to TORA to detect partitions and to merge groups, which seems to be rather heavy for mobile nodes. DAG based algorithms dedicated to unicast routing are described in [8, 4, 30]. We have discussed them intensively in previous sections.

In wired networks, the token circulation is considered as an efficient way to support various distributed algorithms [2, 43, 27, 36, 29]. Therefore, researchers in ad hoc networks tend to extend it to mobile environments. Dolev et al. [7] consider large scale networks with random mobility patterns and thus turn to probabilistic approaches, which results in a circulation period of $O(n^3)$. In the scenarios we considered, the mobility pattern is more regular than what is assumed in [7]; therefore, we can have a deterministic protocol that circulates a token in a period linear with n. Malpani et al. [25] propose several algorithms for token circulation in small ad hoc networks. The idea of our TPP-R is actually borrowed from their work. These algorithms are more light-weight than NASCENT. The reason is twofold: (i) all these algorithms are built upon routing and even transport protocols (DSR+TCP) and thus do not take care of link failures and (ii) the network membership is (implicitly) assumed to be static, which excludes the need for membership management. Unfortunately, their approaches do not apply to VAGs. First, relying on on-demand unicast routing and TCP actually results in larger overall complexity (refer to the analysis in Section 5.2). Secondly, membership management is necessary for VAGs since perturbations (e.g., group forming, partition, and merging) do happen. In addition, we realize that pure token circulation protocols do not (at least not efficiently) support all distributed algorithms. We build our NASCENT at the network layer to reduce overhead and integrate token circulation with a DAG to support concurrent execution of various distributed algorithms.

8. CONCLUSION

In this paper, we have discussed several promising applications of mobile ad hoc networks. All these applications involve only small scale networks and require mainly distributed coordination services, rather than pairwise connections. We term the networks implied by these applications Vicinity Ad-hoc Groups (VAGs), which characterize both the small scale and the group-oriented communication paradigms of these networks. The significance and peculiarities of VAGs motivate us to reengineer network layer protocols, in order to achieve both design time and runtime efficiency for the distributed services required by these applications.

Aiming at replacing routing protocols to support applications that involve only group-oriented communications, we propose NASCENT (Network lAyer ServiCE for viciNiTy ad-hoc groups) dedicated to VAGs. NASCENT is the first[6] to integrate a directed acyclic graph (DAG) with token circulation in building a general network layer service that concurrently supports various distributed services. While the DAG is used to route messages to and from a token holder and to bind VAG members together, the circulated token grants temporary control to its holders. The two components greatly facilitate the implementation of distributed algorithms based on NASCENT. The light-weight membership service embedded in NASCENT requires only localized operations, which is a key to making NASCENT feasible in VAGs. NASCENT also includes new algorithms to initialize a token-oriented DAG among an emerging VAG and to recompose the DAG upon the VAGs' merging. These are also crucial functions for coping with the transiency of VAGs.

In order to defend our claim that NASCENT is an efficient network layer service for VAG members, we have given ex-

[6]Combining logical structure with token passing appeared in [36] and was then introduced into ad hoc networks by [41], but these contributions did not consider token circulation.

amples of distributed algorithms that can be built on top of NASCENT and perform both complexity-based analysis and simulations for NASCENT. The examples and analysis show that the complexity of maintaining NASCENT is of the same magnitude as unicast routing protocols, but that NASCENT greatly reduces the complexity of building and running distributed algorithms, compared to unicast and broadcast/multicast routing protocols as well as flooding. Further simulations prove the feasibility of NASCENT, in the sense that it circulates a token stably and timely even when networks undergo topology changes due to node mobility.

In terms of future work, we intend to focus on one of the discussed applications, for which a real implementation of NASCENT will be carried out. We expect to gather more data on the performance of NASCENT in different environments with the implementation and field tests; this will further motivate the deployment of NASCENT.

9. REFERENCES

[1] C. Carter, S. Yi, P. Ratanchandani, and R. Kravets. Manycast: Exploring the Space between Anycast and Multicast in Ad Hoc Networks. In *Proc. of ACM MobiCom'03*, 2003.

[2] J.M. Chang and N. Maxemchuck. Reliable Broadcast Protocols. *ACM Trans. on Computer Systems*, 2(3):251–273, 1984.

[3] K. Chen and K. Nahrstedt. Effective Location-guided Tree Construction Algorithms for Small Group Multicast in MANET. In *Proc. of IEEE INFOCOM'02*, 2002.

[4] M.S. Corson and A. Ephremides. A Distributed Routing Algorithm for Mobile Wireless Networks. *Wireless Networks*, 1(1):61–81, 1995.

[5] S.K. Das, B.S. Manoj, and C. Siva Ram Murthy. A Dynamic Core Based Multicast Routing Protocol for Ad Hoc Wireless Networks. In *Proc. of ACM MobiHoc'02*, 2002.

[6] X. Défago. Distributed Computing on the Move: From mobile computing to cooperative robotics and nanorobotics. In *Proc. of ACM POMC'01*, 2001.

[7] S. Dolev, E. Schiller, and J. Welch. Random Walk for Self-Stabilizing Group Communication in Ad-Hoc Networks. In *Proc. of IEEE SRDS'02*, 2002.

[8] E. Gafni and D. Bertsekas. Distributed Algorithms for Generating Loop-Free Routes in Networks with Frequently Changing Topology. *IEEE Trans. on Communications*, 29(1):11–18, 1981.

[9] J.J. Garcia-Lunes-Aceves. Loop-Free Routing Using Diffusing Computations. *IEEE/ACM Trans. on Networking*, 1(1):130–141, 1993.

[10] R.S. Gray, G. Cybenko, D. Kotz, and D. Rus. Mobile agents: Motivations and State of the Art. In Jeffrey Bradshaw, editor, *Handbook of Agent Technology*. AAAI/MIT Press, 2002.

[11] Z.J. Haas and B. Liang. Ad Hoc Mobility Management with Uniform Quorum Systems. *IEEE/ACM Trans. on Networking*, 7(2):228–240, 1999.

[12] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In *Distributed Systems*, chapter 5, pages 97–145. Addison-Wesley, 2 edition, 1993.

[13] C.-S. Hsu and Y.-C. Tseng. An Efficient Reliable Broadcasting Protocol for Wireless Mobile Ad Hoc Networks. In *Proc. of IASTED NPDPA'02*, 2002.

[14] J.G. Jetcheva and D.B. Johnson. Adaptive Demand-Driven Multicast Routing in Multi-Hop Wireless Ad Hoc Networks. In *Proc. of ACM MobiHoc'01*, 2001.

[15] L. Ji and M.S. Corson. Differential Destination Multicast - a MANET Multicast Routing Protocol for Small Groups. In *Proc. of IEEE INFOCOM'01*, 2001.

[16] D.B. Johnson, D.A. Maltz, and Y-C. Hu. *The Dynamic Source Routing Protocol for Mobile Ad Hoc Networks (DSR)*, April 2003. Internet-Draft, draft-ietf-manet-dsr-09.txt. Work in progress.

[17] C.V. Jones and Maja J. Matarić. Communication in Multi-Robot Coordination. Technical report, USC CRES-04-001, 2004.

[18] M.-O. Killijian, R. Cunningham, R. Meier, L. Mazare, and V. Cahilli. Towards Group Communication for Mobile Participants. In *Proc. of ACM POMC'01*, 2001.

[19] U. Kozat and L. Tassiulas. Network Layer Support for Service Discovery in Mobile Ad Hoc Networks. In *Proc. of IEEE INFOCOM'03*, 2003.

[20] S.-J. Lee, W. Su, and M. Gerla. On-Demand Multicast Routing Protocol in Multihop Wireless Mobile Networks. *ACM/Kluwer Mobile Networks and Applications*, 7(6):441–453, 2002.

[21] J. Liu, K. Sohraby, Q. Zhang, B. Li, and W. Zhu. Resource Discovery in Mobile Ad Hoc Networks. In M. Ilyas, editor, *Ad Hoc Wireless Networks*, chapter 26. CRC Press, 2003.

[22] M. Lott, R. Halfmann, E. Schulz, and M. Radimirsch. Medium Access and Radio Resource Management for Ad Hoc Networks based on UTRA TDD. In *Proc. of ACM MobiHoc'01*, 2001.

[23] W. Lou and J. Wu. Double-Covered Broadcast (DCB): A Simple Reliable Broadcast Algorithm in MANETs. In *Proc. of IEEE INFOCOM'04*, 2004.

[24] J. Luo, P.Th. Eugster, and J.-P. Hubaux. Route Driven Gossip: Probabilistic Reliable Multicast in Ad Hoc Networks. In *Proc. of IEEE INFOCOM'03*, 2003.

[25] N. Malpani, N. H. Vaidya, and J. L. Welch. Distributed Token Circulation in Mobile Ad Hoc Networks. In *Proc. of IEEE ICNP'01*, 2001.

[26] N. Malpani, J. Welch, and N. Vaidya. Leader Election Algorithms for Mobile Ad Hoc Networks. In *Proc. of ACM DIAL-M'00*, 2000.

[27] N. Maxemchuck and D. Shur. An Internet Multicast System for the Stock Market. *ACM Trans. on Computer Systems*, 19(3):384–412, 2001.

[28] S. Nesargi and R. Prakash. MANETconf: Configuration of Hosts in a Mobile Ad Hoc Network. In *Proc. of IEEE INFOCOM'02*, 2002.

[29] S. Nishio, K.F. Li, and E.G. Manning. A Resilient Mutual Exclusion Algorithm for Computer Networks. *IEEE Trans. on Parallel and Distributed Systems*, 1(3):244–355, 1990.

[30] V.D. Park and M.S. Corson. A Highly Adaptive Distributed Routing Algorithm for Mobile Wireless Networks. In *Proc. of IEEE INFOCOM'97*, 1997.

[31] C.E. Perkins, E.M. Belding-Royer, and S.R. Das. *Ad hoc On-Demand Distance Vector (AODV) Routing*, 2003. RFC 3561 (draft standard). IETF.

[32] C.E. Perkins and P. Bhagwat. Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers. In *Proc. of ACM SIGCOMM'94*, 1994.

[33] R. Prakash and R. Baldoni. Architecture for Group Communication in Mobile Systems. In *Proc. of IEEE SRDS'98*, 1998.

[34] J.K. Redi. Wireless Networking for Mobile Robots. In *Invited talk for Winlab/Berkeley FOCUS'99*, 1999.

[35] D. Reichardt, M. Miglietta, L. Moretti, P. Morsink, and W. Schulz. CarTALK 2000 - Safe and Comfortable Driving Based Upon Inter-Vehicle-Communication. In *Proc. of IEEE IV'02*, 2002. `http://www.cartalk2000.net`.

[36] G. Ricart and A.K. Agrawala. Author's Response to 'On Mutual Exclusion in Computer Networks' by Carvalho and Roucairol. *Communications of the ACM*, 26(2):147–148, 1983.

[37] G.-C. Roman, Q. Huang, and A. Hazemi. Consistent Group Membership in Ad Hoc Networks. In *Proc. of IEEE/ACM ISCE'01*, 2001.

[38] E.M. Royer and C.E. Perkins. Multicast Operation of the Ad-hoc On-demand Distance Vector Routing Protocol. In *Proc. of ACM MobiCom'99*, 1999.

[39] E.M. Royer and C.-K. Toh. A Review of Current Routing Protocols for Ad Hoc Mobile Wireless Networks. *IEEE Personal Communications*, 6(2):46–55, 1999.

[40] A. Schiper. Failure Detection vs Group Membership in Fault-Tolerant Distributed Systems: Hidden Trade-Offs. In *Proc. of PAPM-ProbMiV'02, LNCS 2399*, 2002.

[41] J.E. Walter, J.L. Welch, and N.H. Vaidya. A Mutual Exclusion Algorithm for Ad Hoc Mobile Networks. *Wireless Networks*, 7(6):585–600, 2001.

[42] K.H. Wang and B. Li. Efficient and guaranteed service coverage in partitionable mobile ad-hoc networks. In *Proc. of IEEE INFOCOM'02*, 2002.

[43] B. Whetten, T. Montgomery, and S. Kaplan. A High Performance Totally Ordered Multicast Protocol. In *Theory and Practice in Distributed Systems, LNCS 938*, 1994.

[44] J. Yoon, M. Liu, and B. Noble. Random Waypoint Considered Harmful. In *Proc. of IEEE INFOCOM'03*, 2003.

APPENDIX. PSEUDOCODES FOR TOKEN PASSING PROTOCOL (TPP)

We report the two algorithms of TPP together for compactness, but prefix [$\mathbb{Q}$] and [$\mathbb{R}$] to the lines of code dedicated to TPP-Q and TPP-R, respectively. Lines of codes without prefix are common to both algorithms.

	name	functions
TPP-Q	*epoch*	A counter that logs which cycle of token circulation the member is in.
	reqsent	A pointer that directs to another member to which a token request has been sent.
	$\mathcal{Q}$	A queue that stores token requests (*token_req*s) from neighbors.
TPP-R	*recency*	An array that records the time when the token visited each neighbor recently. The information is piggybacked with the token.
both	*tokenheld*	A flag that designates whether the member holds the token or not.

Table 3: Variables used by TPP.

Table 3 shows the variables used by TPP. Each *token_req* is a triple [*saddr*, *epoch*, *arv_seq*], which denotes the *addr* and *epoch* of a requester, as well as a sequence number increased by each new item. Operations on $\mathcal{Q}$ are based on 3 primitives: ENQUEUE(), DEQUEUE(), and DELETE(). ENQUEUE() inserts an item in a queue following a lexicographical order of [*epoch*, *arv_seq*]; a newly inserted item annihilates an existing one from the same *saddr*. DEQUEUE() removes the item at the queue header and returns the corresponding *saddr*. Items can also be removed from a queue by DELETE(). TPP also maintains a timer *token_timer* and sets it whenever the member hands on the token to another member. If it times out before the token returns, the member suspects a partition and attempts to regenerate a token.

There are four events that can trigger actions of TPP. We discuss them one by one.

ViewChg

LMTP generates a ViewChg event when the *lview* is changed (e.g., by any *bsm*s or by the PRever primitive). Fig. 14 shows how TPP responds to this event. The actions are taken only if the request queue is not empty, which excludes the initialization phase but includes the merging phase. If a token holder detects a group merging and has to give up its right to hold the token due to a smaller new *gid*, it stores the token and forwards a request (lines 3–5). Otherwise, any request from a previously incoming but currently missing or outgoing link is removed from the queue (lines 8–9). Then TPP decides if a token request should be rerouted due to the view changes. A request is rerouted if this member does not become a sink and (i) the link between this member and its *reqsent* is either reversed or broken (lines 11–12) or (ii) the priority of the queue, indicated by the request with minimum *epoch*, is changed (lines 13–14). Otherwise the request is rerouted upon next ViewChg resulting from an invocation of the PRever primitive (lines 15–16). Note that all operations related to the request queue are triggered only for TPP-Q.

```
1:  upon ViewChg do
2:    if |Q_i| ≠ ∅ then
3:      if tokenheld_i ∧ (token.gid > lview_i.gid) then
4:        tokenheld_i ← false; Store(token)
5:        [Q] FwdReq(ture)
6:      else
7:        [Q] priority ← min_{req∈Q_i}{req.epoch}
8:        for all nid ∉ lview_i and
              all nid ∈ lview_i s.t. nid < id_i do
9:          [Q] Delete(Q_i, nid.addr)
10:       if id_i > min_{nid∈lview_i}{nid} then
11:         if (id_i < reqsent_i) ∨ (reqsent_i ∉ lview_i) then
12:           [Q] FwdReq(true)
13:         else if priority ≠ min_{req∈Q_i}{req.epoch} then
14:           [Q] FwdReq(false)
15:       else
16:         [Q] reqsent_i ← id_i

17: procedure FwdReq(redirect)
18:   if redirect then
19:     reqsent_i ← min_{nid∈lview_i}{nid}
20:   token_req.saddr ← addr_i
21:   token_req.epoch ← min_{req∈Q_i}{req.epoch}
22:   Send(token_req) to reqsent_i.addr
```

Figure 14: Response to ViewChg in TPP of member i

```
1:  upon Recv(token_req) from id_j do
2:    if (|Q_i| ≠ ∅) ∧ (id_i < id_j) then
3:      [Q] priority ← min_{req∈Q_i}{req.epoch}
4:      [Q] Enqueue(Q_i, token_req)
5:      if priority ≠ min_{req∈Q_i}{req.epoch} then
6:        [Q] FwdReq(false)
7:      if ∄nid ∈ lview_i s.t. nid = id_j then
8:        Insert(lview_i, id_j)              /* incur ViewChg */
```

Figure 15: Response to the reception of a token request in TPP of member i

Recv*(token_req)*

The reception of a *token_req* triggers actions of TPP-Q only if the member is not in the initialization phase and the requester has an outgoing link towards this member, as shown in Fig. 15 (line 2). The newly received request is put into the queue, and the request is forwarded if the priority of the queue is changed by this request (lines 3–6). The reception of messages other than *bmsg*, such as *token_req* and *token*, is also considered as an input of LMTP (lines 7–8).

Recv*(token)*

If there is a token stored locally (as a result of group merging, see Fig. 14 line 4), the member merges the two tokens (Fig. 16 lines 2–3). While TPP-R decides if to deliver the token according to local recency information after updating its *recency* array, TPP-Q delivers the token only if it is the first in the queue, otherwise it forwards the token and a request (lines 4–15).

```
1:  upon Recv(token) from id_j do
2:    if IsStoreToken() then
3:      MergeToken(token)
4:    [R] recency_i ← token.recency      /* for neighbors only */
5:    if recency_i[id_i] has the minimum value in recency_i then
6:      [R] tokenheld ← true
7:      [R] Deliver(token)               /* to the upper layer */
8:      [R] recency_i[id_i] ← current time
9:    else
10:     [R] Release(token)
11:   [Q] reqsent_i ← getIDinView(lview_i, Dequeue(Q_i))
12:   if reqsent_i = id_i then
13:     [Q] tokenheld ← true
14:     [Q] Deliver(token)               /* to the upper layer */
15:   else
16:     [Q] Send(token) to reqsent_i.addr; FwdReq(false)
17:   if ∄nid ∈ lview_i s.t. nid = id_j then
18:     Insert(lview_i, id_j)            /* incur ViewChg */
```

Figure 16: Response to the reception of the token in TPP of member i

Release*(token)*

This event occurs when the upper layer algorithms finish using the token and return it to NASCENT. TPP-R simply gives the token to the least recently visited neighbor and piggybacks the recency information (Fig. 17 lines 2–3). TPP-Q passes the token to the requester whose request is the first in the queue. Then it changes its state variables and forwards a token request for the next cycle. In both cases, the *token_timer* is set in order to detect group partitions (lines 4–10). Upon *token_timer* times out, a member starts a new initialization phase with a smaller *gid* (Fig. 17 lines 11–14).

```
1:  upon Release(token) do
2:    [R] token.recency ← recency_i
3:    [R] Send(token) to nid.addr
          s.t. recency_i[nid] has the minimum value in recency_i
4:    [Q] reqsent_i ← getIDinView(lview_i, Dequeue(Q_i))
5:    [Q] Send(token) to reqsent_i.addr
6:    [Q] tokenheld_i ← false; epoch_i ← epoch_i + 1
7:    [Q] req.saddr ← addr_i; req.epoch ← epoch_i
8:    [Q] Enqueue(Q_i, req)
9:    [Q] FwdReq(false)
10:   token_timer_i ← 0

11: upon Timeout(token_timer_i, τ_part) do
12:   init_i ← true; init_timer_i ← 0
13:   lview_i.gid.β ← lview_i.gid.β − 1
14:   lview_i.gid.addr ← addr_i
```

Figure 17: Response to the release of the token in TPP of member i