A Tuple Space for Social Networking on Mobile Phones

Emre Sarigöl Oriana Riva Gustavo Alonso

Systems Group, Department of Computer Science, ETH Zurich
8092 Zurich, Switzerland
{emres,oriva,alonso}@inf.ethz.ch

***Abstract*— Social networking is increasingly becoming a popular means of communication for online users. The trend is also true for offline scenarios where people use their mobile phones to network with nearby buddies. In this paper, we propose a distributed tuple space for social networking on ad hoc networks. We describe the tuple space model and its operations and give evidence of its advantages for ad hoc social networking through several applications implemented on top of the tuple space.**

I. Introduction

Today people use their phones in ad hoc mode mostly to establish point-to-point communications to synchronize applications, share contact information, or connect to GPS receivers. In the near future, ad hoc networks will expand to more complex mesh topologies and embrace a wider range of social applications such as those today supported by social network portals. This trend is confirmed by offerings such as MobiLuck [1], ProxiDating [2] or Jambo Networks [3] that have shown a growing interest of users in establishing ad hoc social networks with people in their proximity.

Wireless ad hoc networks fit this type of service particularly well because they can be created on-the-fly and without infrastructure support, for instance between people meeting in the street or on a train. However, even in small scale configurations, building and operating applications in ad hoc networks is a challenging task. The complexity arises from the lack of any fixed infrastructure or centralized administration, and the unstable communication topology [4], [5].

In this paper, we are concerned with storing and sharing data in ad hoc networks as a fundamental service over which a large variety of applications can be built. Existing solutions to data storage and lookup in ad hoc networks exhibit at least one of the following shortcomings:

- Lack of flexibility in specifying the semantics of the search query in terms of reachability [6]
- Lack of support for disconnected operation [7]
- Dependence on a specific routing protocol [8], [9]
- Reliance on the presence of a fixed and stable super node [10] or of an overlay network that cannot be maintained without high cost [11], [12]
- Need to occasionally flood the network with data advertisements and lookup messages [13]
- Reliance on geographical information [14], [10]

We present a novel approach to storing and searching data in ad hoc networks that overcomes these limitations. The key insight is to provide the abstraction of a distributed tuple space in ad hoc networks, which embodies database and messaging capabilities in a unique model, accessible through a simple, yet powerful API. The tuple space abstracts the underlying network as the common memory space in which nodes can store and lookup key/value pairs (i.e., tuples). However, compared to traditional tuple spaces, in our model each application configures its own "shared memory" rather than all tuples residing in a single shared memory. Moreover, an application can control the propagation of its tuples both in space and time. Tuple insertion is controlled by tuning the *scope* of the dissemination process, and the *version* and *lifetime* of each tuple. Lookup of tuples occurs through partial or full matching of keys and values, while the communication channel features both synchronous client-server messaging and asynchronous publish-subscribe. Tuples inserted in the tuple space disappear once their lifetime expires or upon an explicit deletion operation.

II. Tuple space model

A tuple consists of a sequence of typed fields. Besides containing a *key* and a *value*, as in traditional tuple space models [15], [16], it also has an *owner*, a *scope*, a *lifetime*, and a *version* field. The key is the identifier of the tuple and the value represents its content. The owner identifies the node generating the tuple. The scope field specifies for how many hops at most the tuple must be disseminated in the network. The lifetime field specifies for how long a tuple is stored locally at a node. Finally, the version field specifies a replacement schema among tuples with the same key. Therefore, a tuple is uniquely identified by the pair $< key, version >$.

A presence application like the one described in Section IV, uses a tuple like the following to advertise the name of a buddy, its status, and IP address.

```
tuple-presence = {
  key = "adsocial-presence"
  value = "15.10.5.2:80|Patrick|busy"
  version = 32
  owner = "15.10.5.2"
  scope = 4
  lifetime = 30
}
```

Our tuple space provides primitives for insertion, deletion, and lookup of tuples. We describe these operations in the

following.

A. Tuple dissemination

Traditional tuple space usually rely on full replication of the tuples on all nodes participating in the space. This approach cannot work in our context because it is too demanding for mobile phones in terms of memory and communication resources. However, even if full replication could be guaranteed, it is not a practicable solution if node disconnections are frequent. Instead, we let application developers to decide the degree of replication of their tuples and hence the level of consistency that can be guaranteed. A tuple is inserted into the tuple space through the *put(tuple)* operation and by tuning the scope and lifetime parameters it is possible to customize the degree of replication of such a tuple.

B. Query and tuple template

Lookup of tuples stored in the network occurs through the *get(query)* and *scan(query)* operations. They both search and return tuples matching a given tuple template, but they differ in the way the query is disseminated and the range of returned results. The *get(query)* operation propagates the input query in the network until one or more matching tuples are found, whereas *scan(query)* returns all matching tuples stored on all nodes in the specied search space.

Both get and scan take a query as input. In our tuple space everything is managed as a tuple and thereby queries are also wrapped into a tuple structure with special semantics. The matching criteria is specied using the key, version, and, optionally, the value field. These parameters dene a sort of tuple template. The owner, scope, and lifetime of the query are instead used to identify the node submitting the query, and the geographical (scope) and temporal (lifetime) boundaries of the query dissemination process. Unlike tuples, queries are uniquely identied by the four parameters $< key, value, version, owner >$. This identication allows to decide when to store and answer a query, and prevents a node from unnecessarily processing or forwarding the query.

An example of query for the above presence-tuple is the following.

```
query-presence= {
  key = "adsocial-presence"
  value = *
  version = LAST
  owner = "15.10.5.3"
  scope = 5
  lifetime = 100
}
```

For queries, $*$ and $LAST$ has specic meanings when used with some of the tuple field. The former expresses the any relationship and can be used in the key, value, and version elds of a query. The latter is used only in the version field of queries to retrieve the most up-to-date tuples from each visited node. $LAST$ allows to reduce the number of tuples returned and it is particularly useful in the case of tuples with a long lifetime of which many versions are likely to exist.

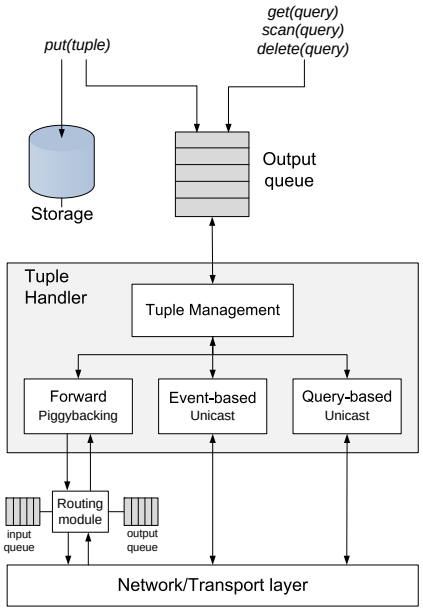


Fig. 1. Tuple space architecture.

C. Tuple lifecycle

Atomicity is a basic feature of the stateful model of traditional tuple spaces. Other tuple space systems have solved this issue by adding costly, transaction-based mechanisms [6], [17] which are too resource-demanding and hardly feasible in ad hoc networks. We address the problem by associating a lifetime and version to every tuple. When the lifetime of a tuple expires, it is automatically removed from the tuple space. This provides two advantages over traditional systems: first, once a tuple is distributed, its removal from the tuple space does not require additional cost, which is optimal for ad hoc networks. Second, such a "garbage collection" mechanism works naturally under frequent node disconnections, node failures and network partitions. In the case of queries, the lifetime parameter is particularly important because it allows to support both one-time queries ($lifetime = 0$) and long-term subscriptions ($lifetime > 0$) for local or remote events.

Although the lifetime field guarantees that no stale tuple will remain in the system, an application may still need to explicitly delete tuples. For tuples with a short lifetime, one may simply wait for its lifetime to expire, or insert an update of the same tuple with an incremented version. However, we provide also an explicit delete operation that is particularly useful for removing long-term subscription queries before their lifetime expires.

III. Tuple space implementation

The tuple space is provided by a distributed system that runs as a user space process on various nodes in the network. Figure 1 shows the system architecture. Every node has a local storage where generated or received tuples/queries are kept. Outgoing queries and tuples are placed in an output queue and then passed to the *Tuple Handler*. This processes incoming

messages by performing tuple matching and updating tuple parameters, and dispatches them to the appropriate communication channel. Communication channels are of three types: forwarding by piggybacking tuples on routing messages, event dispatching for asynchronous communication, and unicast messages for client-server communication. In the following we describe in more details how these modules interact to accomplish tuple propagation and query processing.

A. Tuple and query dissemination

Tuple dissemination occurs by piggybacking tuples on routing messages exchanged by the underlying routing protocol. We use the piggybacking approach of the MAND platform that also allows to support services like SIP, DNS, and SLP in ad hoc networks [18]. A so-called *MAND routing handler* constantly listens for incoming routing messages and is capable of piggybacking tuples on an outgoing routing message or extracting tuples from a received one.

The piggybacking-based approach provides significant advantages over existing solutions. First, the piggybacking module does not require modification of the routing protocol with which it works. Second, this approach is highly message-efficient since it does not trigger any additional traffic. However, since the tuple propogation rate is limited by the availability of incoming or outgoing routing packets, unicast messages are used in cases where timely delivery is necessary.

We optimize the tuple propagation approach of MAND by adding support for tuple aggregation and fragmentation. If more than one pending tuple is present in the output queue, tuples are aggregated onto the next available routing packet until there are no more pending tuples or the routing packet is full. If a tuple exceeding the maximum tuple size has to be sent, this is fragmented into multiple tuples with the same key and an incremented version.

Regardless of the routing protocol employed in the network layer, unless nodes impose a termination condition on tuple retransmissions, tuples (and queries) will be indefinitely transmitted across the network in a "ping-pong" fashion. To prevent this from happening, at the tuple space level we provide specialized filtering mechanisms. Since we know that tuples locally stored on a node have already been forwarded if they have scope$>$0, the node only forwards an incoming tuple if it has a higher version than the local tuple. Queries use the version field as part of tuple matching, therefore filtering of an incoming query is based only on scope: an incoming query is forwarded only if it has a scope larger than the query locally stored.

B. Query processing

Each time a node receives a get, a scan or a delete request, it follows a 2-step process.

In the case of get, if one or more matching tuples are available locally, it returns them to the query issuer. Until the query expires, every new matching tuple will be returned. Only if no matching tuple is available locally and if the scope parameter allows it, the query is further propagated in the network. Before forwarding the query, the node decrements the scope value such that queries are propagated only until their scope becomes 0.

A scan request is processed in the same way, with the only difference that a node continues propagating the query even though matching tuples are found locally. The reason for supporting both get and scan operations is a trade-off between completeness of the results and induced cost. A scan operation is clearly more expensive, but allows to cover a larger search space and potentially return all results. A get operation saves resources by stopping at the first few hits in the network.

Replies to scan and get operations are sent back to the query issuer using unicast messages directed to the IP address specified in the owner field. Delete requests are processed in the same way as scan requests.

C. Deployment

The tuple space platform has been designed and implemented for Linux and it is mainly written in C/C++. The system was deployed by keeping in mind the resource constraints and unique features of mobile devices and it currently runs on Nokia N810 Internet tablets. These devices are small enough to be carried around, they support IEEE 802.11b/g and run the Debian Linux-based Maemo [19]. Ad hoc networks are established using WiFi.

The system has been tested both with controlled lab experiments and in several indoor/outdoor field trials where 30+ users interacted with the system and its applications.

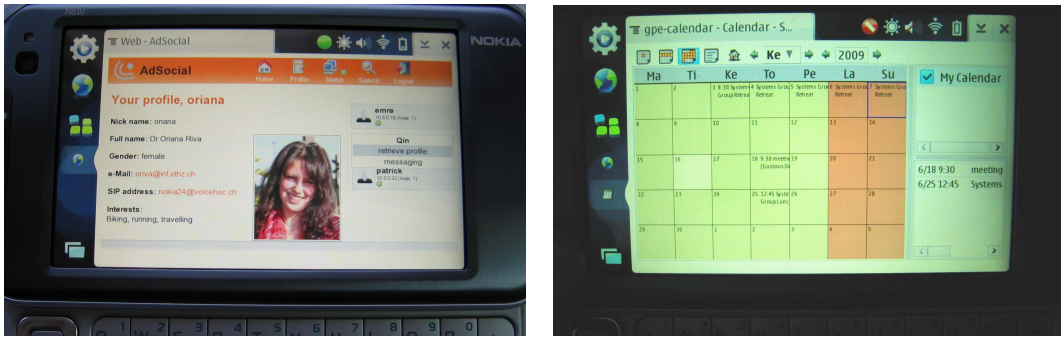
IV. Applications

Using the tuple space architecture we have implemented several applications such as a buddy presence service, a calendar application, video and VoIP communication, and instant messaging [20]. In the following we described two example applications.

A. Buddy presence

We have used the tuple space to implement a buddy presence service (shown in Figure 2(a)) that allows users to view all buddies in their proximity as well as search for buddies with specific interests. Such an application could be used, for instance, in a conference venue to allow people to set up discussion groups. Every user creates a profile that among several personal information contains a list of interests. A conference attendee may, for example, express in his profile an interest for "social networks & information filtering". If another conference participant wishes to meet with researchers working in his field, he can initiate a search with some relevant keywords and retrieve a list of users with matching interests.

Unlike in online social networks running in reliable networks, in ad hoc social networks, a buddy's presence status and interests must be periodically updated in the network. This happens by piggybacking on routing messages. When a buddy search is initiated the tuple space places a query in the output queue.z The query is first evaluated by matching tuples stored in the local tuple space. If no matching tuple is found or a more



(a) Buddy presence service

(b) Calendar application

Fig. 2. Examples of applications built using the tuple space architecture running on a N810 handheld.

accurate search is required the query is propagated across the entire social network.

B. Calendar application

Our tuple space currently supports also a calendar application shown in Figure 2(b). This application particularly benefits from the flexible API our tuple space provides. Each time a user generates a new calendar entry or updates an existing one, an update tuple is propagated into the network. Users can subscribe to receive all updates or only some relevant ones. For example, filters on the key of a subscription tuple may be associated to room number, project name, or names of participants. A user can also retrieve the entire history behind an updated calendar entry and receive information about all users who have modified such an entry in the past by specifying lookup tuples for certain (e.g., $version = LAST$, $version = 3$) or all ($version = *$) versions of a calendar entry.

V. Conclusions

This paper has presented a tuple space system capable of supporting services typical of online social networks, but in offline mode, through multi-hop ad hoc networks of mobile phones. The system is both an in-memory data storage and a communication system with a simple and flexible API. Compared to the current state-of-the-art in building data storage and lookup systems in ad hoc networks, our system provides flexibility, is compatible with a large number of existing routing protocols, and is highly message-efficient. We have shown how applications can use the tuple space and we gave evidence of its effectiveness with several real-world applications built using it.

Acknowledgment

We thank Patrick Stuedi for his support during the development of the tuple space system.

References

[1] "MobiLuck," 2009, http://www.mobiluck.com.
[2] "Proxydating," 2009, http://www.proxidating.com.
[3] "Jambo Networks," 2009, http://www.jambo.net/.
[4] P. Gupta and P. R. Kumar, "The capacity of wireless networks," *IEEE Transactions on Information Theory*, vol. 46, pp. 388–404, 2000.
[5] P. Gupta and P. R. Kumar, "Critical power for asymptotic connectivity in wireless networks," *Stochastic Analysis, Control, Optimization and Applications*, pp. 547–566, 1998.
[6] A. L. Murphy, G. P. Picco, and G.-C. Roman, "LIME: A Middleware for Physical and Logical Mobility," in *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS'01)*. IEEE Computer Society, May 2001, p. 524.
[7] M. Mamei and F. Zambonelli, "Programming Pervasive and Mobile Computing Applications with the TOTA Middleware," in *Proceedings of the 2nd IEEE International Conference on Pervasive Computing and Communications (PerCom'04)*. IEEE Computer Society, 2004, p. 263.
[8] H. Puch, S. Das, and C. Hu, "Ekta: An efficient DHT Substrate for Distributed Applications in Mobile Ad Hoc Networks," in *Proceedings of the 6th IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'04)*, December 2004, pp. 163–173.
[9] T. Zahn and J. Schiller, "MADPastry: A DHT Substrate for Practicably Sized MANETs," in *Proceedings of the 5th Workshop on Applications and Services in Wireless Networks (ASWN'05)*, June 2005.
[10] F. Araujo, L. Rodrigues, J. Kaiser, and C. Liu, "CHR: a distributed Hash Table for Wireless Ad Hoc Networks," in *Proceedings of the 4th International Workshop on Distributed Event-Based Systems (DEBS'05)*, June 2005, pp. 407–413.
[11] C. Cramer and T. Fuhrmann, "Proximity Neighbor Selection for a DHT in Wireless Multi-Hop Networks," in *Peer-to-Peer Computing*, 2005, pp. 3–10.
[12] R. Kummer, P. Kropf, and P. Felber, "Distributed Lookup in Structured Peer-to-Peer Ad-Hoc Networks," in *OTM Conferences (2)*, 2006, pp. 1541–1554.
[13] K. Patel, S. Iyer, and K. Paul, "RINGS: Lookup Service for Peer-to-Peer Systems in Mobile Ad Hoc Networks," in *Proceedings of the 6th International Workshop on Distributed Computing (IWDC'04)*, no. 3326. LNCS, December 2004, pp. 471–476.
[14] J. B. Tchakarov and N. H. Vaidya, "Efficient Content Location in Wireless Ad Hoc Networks," in *Proceedings of the 5th IEEE International Conference on Mobile Data Management (MDM'04)*. IEEE Computer Society, January 2004, pp. 74–85.
[15] D. Gelernter, "Generative communication in Linda," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 7, no. 1, pp. 80–112, 1985.
[16] T. J. Lehman, A. Cozzi, Y. Xiong, J. Gottschalk, V. Vasudevan, S. Landis, P. Davis, B. Khavar, and P. Bowman, "Hitting the distributed computing sweet spot with tspaces," *Computer Networks*, vol. 35, no. 4, pp. 457 – 472, 2001.
[17] C. Julien and G.-C. Roman, "EgoSpaces: Facilitating Rapid Development of Context-Aware Mobile Applications," *IEEE Trans. Softw. Eng.*, vol. 32, no. 5, pp. 281–298, 2006.
[18] P. Stuedi, M. Bihr, A. Remund, and G. Alonso, "SIPHoc: Efficient SIP Middleware for Ad Hoc Networks," in *Proceedings of Middleware'07*, ser. LNCS, vol. 4834. Springer, 2007, pp. 60–79.
[19] "Maemo Linux," 2009, http://www.maemo.org.
[20] E. Sarigoel, O. Riva, P. Stuedi, and G. Alonso, "Enabling social networking in ad hoc networks of mobile phones," *Demonstration at the 35th International Conference on Very Large Data Bases (VLDB'09)*, 24–28 August 2009.