Opportunistic Content Sharing Applications

Jörg Ott
Department of Communications and Networking
Aalto University, Helsinki, Finland
Jorg.Ott@aalto.fi

Jussi Kangasharju
Helsinki Institute for Information Technology
University of Helsinki, Helsinki, Finland
Jussi.Kangasharju@helsinki.fi

ABSTRACT

Opportunistic communication between mobile nodes allows for asynchronous content sharing within groups. Limiting the spread of information to a geographic area creates an infrastructure-less variant of digital graffiti, a social network with coupling in space and limited decoupling in time. Due to its nature, this kind of a communication network lends itself readily to name-oriented abstractions. In this paper, we extend our previous work on floating content, extract its fundamental characteristics, and define a system model and a simple API with a set of basic programming elements to support applications in leveraging opportunistic content sharing as a generic communication facility. We validate our API through application examples and show how their communication needs are mapped to our model. In addition, we also implement our API in our simulator and demonstrate the feasibility of these kinds of applications.

Categories and Subject Descriptors

C.2 [**Computer-Communication Networks**]: Network Architecture and Design; D.1 [**Programming Techniques**]: Miscellaneous

General Terms

Design, Experimentation

Keywords

opportunistic content sharing, application programming interface

1. INTRODUCTION

Location- and context-aware services are popular for mobile users, in order to find their way around (using, e.g., Google maps and location-related searches) or to share their whereabouts and related information with peers in their online social networks (e.g., FourSquare, Dopplr). An issue gaining attention is the loss of location privacy when location-related inquiries are sent to a server-based system, be it for tiles of a map, for restaurant recommendations, or gas station locations, especially when those use the same service infrastructure. While some systems have been suggested for using diverse pseudonyms when talking to server infrastructure (e.g., [13]) and for using local caching (e.g., [1]), other system designs avoid using a server infrastructure altogether.

One class of such applications provides location-tagged content sharing as the server-less counterpart of digital graffiti [3]: Examples include our work on *Floating Content* [11,15,21], but also *hovering information* [25], Locus [24], and variants targeted at VANETs [17, 19]. While varying in operational details and protocols, these systems share the property that applications *post* content items destined for nodes in a certain *anchor zone* that is specified by an *origin location* and, e.g., a *radius*. An item may be associated with a time-to-live and other parameters. Inside the *anchor zone* mobile nodes encountering each other will replicate a content item to keep it available, outside it may/will be deleted.

At their core, these systems can roughly be considered *data-oriented* or *information-centric* since some nodes *publish* items using some identifier to which other nodes can *subscribe*, thus offering pub/sub-style semantics [12, 14, 16]. While content items can be tagged with explicit metadata (e.g., names [12, 16] or channels [18]) they always carry their respective anchor zone as implicit subscription filters.

In terms of naming, these systems have essentially *three kinds of names* for content items. First, at the lowest level, the item is identified by its origin location and radius, i.e., the name is a physical location in the real world. Second, our API (see Section 4) encapsulates these low-level names and presents *a label* to the application. As detailed in Section 4, we create the labels via hashing over the key properties of the content item. Third, and additional, application-specific naming is provided by the metadata associated with a content item. The metadata defines a tuple space which provides applications a means of filtering content. In this paper, we do not consider the details of such a tuple space, but concentrate instead on the first two kinds of names and their mapping.

Applications use the (name-oriented) labels provided by our API, which "maps" them to the lowest level names, or physical locations. Conceptually, this is similar to a locator/identifier split, but with the catch that such a mapping is feasible only when the node is physically present in the anchor zone of the content item the application wants to access. This presents challenges for designing applications.

A system exclusively based on opportunistic encounters between mobile nodes yields a "best effort" content sharing paradigm: an application posting an item won't know if or how long the item will stay around and how many other nodes will be reached. This poses interesting challenges for designing applications on top of such a system: while plain content sharing may simply remain best effort—originators (e.g., of ads) might not care too much about

NoM'12, June 11, 2012, Hilton Head Island, SC, USA.

who did or did not get a copy and receivers won't know what they missed—other forms of communication are more difficult to satisfy—such as leaving a note for a person or a group or sending a request and expecting a response.

In this paper, we focus on the general properties of this communication paradigm and its use. We do this by defining a common system model and an API suitable for different kinds of applications and provide some basics for programming applications. We aim at keeping the API as simple as possible, yet general enough to support a variety of applications using this paradigm. We validate our model through different example applications that use opportunistic content sharing and show how their widely differing communication requirements can be implemented with our API.

2. BACKGROUND

Communication paradigms can be classified how they are (de)-coupled in space and time. TCP connections over the Internet are coupled in time, but decoupled in space, whereas classical MANETs are coupled in both. Publish/subscribe systems are decoupled in both. The above opportunistic content sharing systems are *coupled in space, but decoupled in time*, making them a separate class of applications from the existing ones. They bear similarities to publish/subscribe applications, but because of the coupling in space (i.e., physical location), are restricted in terms of what network resources they can use. An Internet-based publish/subscribe system can exploit spatial decoupling and store content anywhere in the network (subject to being able to retrieve it later).

A fair amount of research effort has recently gone into APIs for related types of communications. An API for general publish/subscribe systems has been suggested by Demmer *et al.* [5]. They mostly discuss well-connected networks and, while mentioning delay-tolerant and opportunistic communication, they do not put much emphasis on the specifics. Systems and API specifically designed for opportunistic content distribution include, e.g., [8,18], but they don't provide much insight on generic applications. A number of authors address the problem of finding content in opportunistic networks [9,10,23], without much consideration of API issues.

Since surrounding mobile users may be viewed as a cloud of resources to be exploited for computation and retrieval, mechanisms and languages have been developed for spreading computations across mobile nodes [20]—somewhat similar to data center operation with a Map/Reduce approach where a master assigns tasks to workers—and for declarative programming in such an environment [26]. The performance when spreading computation requests in an opportunistic fashion to mobile nodes, including estimating the number of workers to be assigned to a given task to achieve a certain success probability, has been studied in [20,22].

3. SYSTEM MODEL

We choose a simple system model for *Floating Content* that is general enough to approximate also the other above opportunistic content sharing approaches. Node S creates a data item B at a given location $L = (x, y[, z])$. B is associated with two radii, r and a, that define the distance from L in which B will be replicated and valid (i.e., will be kept), respectively; and with a lifetime T after which B will be deleted. Items will also be deleted outside this anchor zone. The associated metadata, β, supports interest-based filtering and application-specific features, e.g., security (see Sec. 8).

The system offers one communication primitive: posting content into an area from which it may be picked up by other nodes. The operation is inherently best-effort and asynchronous: if all nodes carrying a copy of an item leave the anchor zone, the data item disappears; and there is no upper bound (besides T) within which a data item might reach other nodes. This single primitive implies that items can be created and replicated to other nodes, but not be deleted explicitly on other nodes [15]. Since the system is potentially disconnected, no such guarantees could be given anyway.

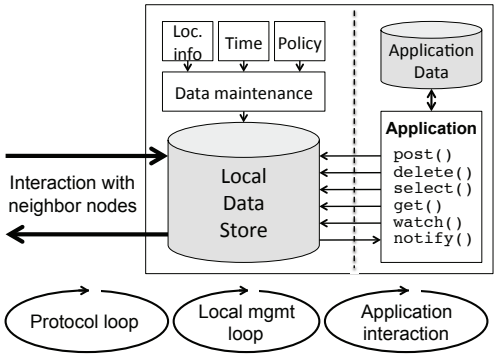

Figure 1: System model

Figure 1 shows the conceptual view of a node. A data store contains items from local applications and other nodes. Three largely independent control loops govern the operation of the data store:

1) The *protocol loop* is responsible for sharing data items with neighboring nodes. The functionality includes peer discovery; negotiating which items to exchange, i.e., determining in which anchor zones both nodes are located; prioritizing the items; and exchanging them reliably. 2) The *local management loop* takes care of deleting items when the node leaves the respective anchor zone or when the item's lifetime expires. It also decides which items to discard if the memory gets full. 3) The *application interaction loop* offers the interface for local applications to the data store and is the focus of this paper.

A newly created item is inserted into the node's local data store, from which the (independent) protocol loop will pick up the item for replication to other nodes. The application posting the item will not know when this replication occurs and how many other nodes will be reached. Items obtained from other nodes show up in the local data store and can be picked up by local applications while inside the anchor zone.

In effect, nodes only communicate via the anchor zone of a given data item, i.e., using physical locations as names. This is conceptually similar to a best-effort blackboard to which posts get written (maybe wiped off), where nodes can view and interact with a limited portion of the board at a time.

This notion also somewhat resembles (scoped) multicasting in the Internet, e.g., in the days of the early Mbone, when distributed state sharing for a class of cooperative conferencing applications—such as *wb* [6] and *nte* [7]—based on *application layer framing (ALF)* [4] were developed. While geographic and multicast scoping can be considered conceptually similar, the major differences are the decoupling in time and the sole reliance on mobile nodes for communication, which make data item delivery more random (in terms of loss, ordering, delay) and retransmissions inherently difficult and thus consistent views harder to achieve.

4. A SIMPLE API

Using the above system model, we present an API plus a set of parameters to be associated with data items in support of diverse

l	$\leftarrow$	*post* $((B,\beta),(L,r,a,T),\langle l\rangle)$
$\emptyset$	$\leftarrow$	*delete* (l)
$\langle(l,\beta)\rangle$	$\leftarrow$	*select* (β,τ)
$((B,\beta),(L,r,a,T),\langle l\rangle)$	$\leftarrow$	*get* (l)
$\emptyset$	$\leftarrow$	*watch* (β,δ,Δ)
$\emptyset$	$\leftarrow$	*notify* $(\langle(l,\beta)\rangle)$

Table 1: Conceptual system API

types of applications. Firstly, table 1 summarizes the signature of the basic methods offered by a geo-based data sharing system. In our API, every data item is associated with a *label* l valid across nodes for space-efficient reference; we specify l to be an item identifier created by a (keyed) hash function over (B,β), (L,r,a,T). A label is a space-efficient, statistically unique reference to a data item and serves two purposes: in the exchange protocol, determining which items to replicate between nodes and, in the application protocol, referring to one or more "previous" items B_i in an item B if B is a semantically dependent on B_i. This allows creating *threads* of data items.Together, the API, local data store and the system loops map the labels to the physical names. All this happens without the application programmer having to worry about it.

post() creates a data item (B,β) for a location L, radii (r,a), and expiry time T and inserts it to the local data store. Upon creation, the system validates that the node is in the anchor zone of the data item. In addition, a vector of item labels $\langle l\rangle$ may be provided to indicate the labels of input items that B depends upon, e.g., because B carries a reaction to one or more other items. This feature allows explicitly linking an arbitrary series of items; the interpretation of linkage is up to the application protocol. Note that such arbitrary series of items form trees, rooted at the "original" item. It is possible to merge different branches of the tree, by creating a new item that depends on the leaves to be merged.

Applications may use the metadata to implement application-specific functionality, e.g. security features such as signatures (see section 8).

delete() is a purely local operation and removes a data item from the local content store, provided that the same application on this node has created the item before. The deleted data item will not be replicated upon future encounters with other nodes; but invoking this method has no bearing on the data stores on other nodes.

select() is a data base-style operation used by an application to synchronously retrieve a set of labels and associated metadata that match the metadata pattern specified in the request. The select call also takes a time parameter τ that specifies how recent the retrieved objects should have been received by the local data store; i.e., in the time window $[-\tau,0]$ before the call was issued, so that applications can iteratively retrieve new items.

get() retrieves the content of a data item and the associated metadata for a given label from the data store. Since nodes move, a label may become invalid between a previous function call, e.g., *select()*, and calling *get()*.

watch() registers application interest in incoming data items matching a metadata spec β with the local data store. δ indicates the minimal interval between notification calls in response to *watch()*. Δ signifies the lifetime of the registration; the latter can also be used to prolong ($\Delta>0$) or terminate ($\Delta=0$) an existing registration.

notify() is a callback-style operation allowing the local data store to inform an application about the reception of new data items matching a previously issued *watch()* by providing a vector of (label,metadata) pairs. The intervals between *notify()* calls adhere to δ, and the system will aggregate notifications. If a data item is deleted from the local data store before the corresponding notification is issued, it won't be included in the notification vector.

5. ITEM-BASED PROGRAMMING

Many practical issues for opportunistic content sharing-based communication can/must be resolved at the application level. Below we sketch some properties that stem from our choice of the system model and API and that are automatically available to any applications using the API. Applications can use the item metadata β to implement additional features. However, all applications must accept the probabilistic best-effort nature of the system's operation. Nodes in a anchor zone may meet in arbitrary order and they may not meet at all. Obviously, creation times of items paired with a sequence of encounters determine how data items spread, i.e., which node receives which items and in which order.

5.1 Ordering and Threads

Data items may be created in response to other data items, yielding *threads*. In an asynchronous multi-sender multicast system, this raises the issue of ordering at a receiver node. Because we cannot guarantee that all nodes will ultimately receive all the items, a notion of a total order does not exist. Therefore, we borrow from the concept of *causal ordering* [2] that suggests including a "vector time" (made up from individual nodes' sequence numbers) to indicate the causal relation between items in a distributed manner. Since an anchor zone is an open system—nodes may arrive and depart at any time—and because unrelated data items may exist in an area, we cannot have a finite vector of sequence numbers. Instead, the predecessors B_i of an item B_n, i.e., those items that were received and have influenced B_n, are explicitly identified in an unordered set: $\{B_i\}\rightarrow B_n$.

If all B_i share a single common root, e.g., B_0, the creation of dependent items yields a directed graph rooted at B_0: nodes having seen different sets of predecessors when creating new items, B_n and B'_n, generate branches in a dependency tree, whereas items created with predecessors from several branches yield merging points.

The set $\{B_i\}$ is supplied by the application and may thus be condensed as appropriate (see section 5.3).

5.2 Anchor Zone and Expiry

When a newly created item is linked to a previous, we require the anchor zones of the two items to be identical. If this was not the case, responses could "wander off" from the original request location and spread arbitrarily as nodes move over time, which would make it impossible for a node to collect them. Our choice guarantees that a set of nodes connected (e.g., in a mesh) in an anchor zone will be able to interact in a consistent manner.

The choice of the expiry time is up to the application. For a new item (B',β'), the expiry time T' may be $T'=T$ so that all state related to the original item disappears at the same time. Choosing $T'>T$, e.g., with the same relative time-to-live, would allow applications to construct a sliding window over a set of items.[1] In the latter case, however, predecessor vectors may include item labels no longer available. It is up to the application to ensure that the necessary state gets carried over, e.g., by generating items that subsume earlier ones, and removing the stale labels from the vector.

5.3 Subsuming Items

To support applications in which a node is responsible for a piece of state that gets updated as per items from other nodes, we allow

[1]There is nothing preventing the choice of $T'<T$, although we do not have a good example where this would be desirable.

a node to issue an item *subsuming* earlier ones, provided that the same node issued the subsumed items (see also section 8). Subsuming means that the newly created item replaces the content of the subsumed ones; this could be done by *overriding* the previous content or *aggregating* data of the input items. This will not affect item replication but is solely meant as an indication to the application and therefore be handled entirely at the application layer. An application should delete subsumed items from the local data store and only replicate the subsuming item.

As a special case, we allow any node to subsume any item, provided that it includes the entire predecessor items (including metadata) in a newly created item, i.e., adding redundancy to the system. One special case of subsuming is open to any node for any item. An application may include the entire predecessor items (including metadata) in a newly created one and thereby add redundancy to the system.

The degree of redundancy (how many and which previous items) can be tailored to meet the reliability demands of the application, for example, using the expiry time window mechanism (section 5.2).

5.4 Consistency Considerations

With the above tools, we can turn our attention to consistency for applications. Due to the disconnected and asynchronous operation, no general guarantees can be made for availability or delivery of any individual item B.

Consider a single anchor zone of an item B_0. As nodes may enter and leave an area arbitrarily[2], we first consider a stable group of n nodes that stay inside an anchor zone sufficiently long. In a non-congested system with sufficient node density, each of the n nodes should be able to obtain a copy of B_0 after a random delay. As corollary, if any two nodes X and Y remain inside the anchor zone, they should eventually obtain all items in this area. By definition (section 5.2), this includes B_0 and all its descendants B_i, i.e., direct or indirect responses to B_0 or subsuming items—as long as each item's residual lifetime upon creation is sufficiently large to reach X and Y.[3] We can also look at the system from *an item's point of view*. As nodes may post responses to items or subsume old items, we obtain chains and trees of items dependent on each other. To obtain a complete view of the state and history of the item, we need to obtain all these dependent items as well. Our assumption about the content area of linked items being equal to the original items (see Section 5.2), implies that the whole history of the item is available in the same anchor zone (but with potentially different expiry times T). Again, assuming a sufficient node density and sufficiently long expiry times, all copies of the item in the anchor zone will have received all the new, dependent items, thus leading to *eventual consistency*. It is up to the application protocol to prevent or deal with state inconsistencies arising from nodes receiving items in different orders and responding to different subsets.

Because of the probabilistic nature of the system, attaining eventual consistency is the best case scenario. In reality, items may disappear because of too low node densities, too short expiry times, etc.; nodes may arrive late and see only part of the state or leave early. This implies that, in general, applications cannot assume any kind of consistency and need to deal with nodes seeing (different snapshots of) different branches of the directed item dependency graph. Application protocols may support the best-effort operation by a number of mechanisms, such as 1) including by earlier items redundantly in new ones, 2) idempotent posts of full state (rather than deltas), 3) defining commutative and independent operations, as we will show next by example.

[2]Entering and leaving an anchor zone is somewhat similar to joining and leaving a multicast group, with a multicast service featuring losses, highly variable delays, and asymmetric paths.

[3]This suggests using the sliding window approach above.

6. APPLICATIONS

As noted above, we consider the existing work [3, 17, 19, 21, 24, 25], as instances of the same opportunistic content sharing paradigm. Each particular system has its own protocol and local management loops (Fig. 1), but their respective properties allow supporting our API. These systems suggest diverse application scenarios, mostly based on one-to-many content sharing without much elaboration on more sophisticated interactions. We now present different application examples, partly inspired by the above, and show how their communication needs can be satisfied by our simple API.

Graffiti: The simplest application is essentially digital graffiti, where some users post items and others read them. Each item is self-contained, with no dependencies between items. The application calls *post* to create new items and reading happens with *select* and *get*. Note that the operation is fully pull-based. Since all items are stand-alone, there is no need for ordering, subsuming, and consistency functions.

Collaborative sensing is a variant of graffiti: Consider a fully distributed sensing application, e.g., mapping of WLAN access points. Devices *post* information about discovered APs, e.g., MAC address, SSID, signal strength, etc., and implicitly their own location, using an anchor zone radius well exceeding the AP coverage. *select* and *get* allow nearby nodes to learn about WLANs (possibly filtered by SSID as metadata); by combining origin information from different items, they can infer an AP's approximate location. An application uses *watch* to ask for a *notify* when it gets close to certain APs. While items from different nodes are independent and no ordering or dependencies exist, a device may subsume its previously posted items with new readings, and different devices may repeat other items for robustness.

Regional chat: In a chat application users *post* messages and set appropriate filters with *watch*, e.g., matching their pseudonyms. When new messages appear, *notify* will be triggered and the application can offer the user to *post* replies. Since all communications are targeted to specific users, *select* and *get* are not needed because *notify* will ensure that the application gets a chance to react to new messages. Chat would use pointers to predecessors to show dependencies and likely include a limited number of previous items redundantly in new *post*s. Different threads would start from different roots and form different dependency graphs.

Local auction: On a flea market users *post* about the (real-world) items they sell. Buyers use *select* to find items of interest and *get* for more information about an item. They can also *watch* for new items coming on sale as a passive search and then react to a *notify*. In either case, a user can *post* a reply, e.g., a bid for an item. The seller uses *watch* and *notify* to collect the bids and can *post* messages about ending the auction. Item dependencies only exist between item offers, bids, and auction closures, which can be modeled using predecessors; item offers may be updated (subsuming earlier ones) indicating the currently highest bid received. Buyers closer to a seller may have better chances to win—as bids from further away get delayed or lost—but this just reflects real life. Here, a central node (seller) collects state information (bids) from the network: we have multiple sources and one sink as in some classes of wireless sensor networks, which thus could also be supported by our model.

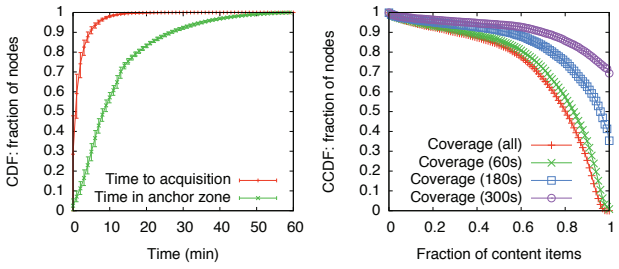

Figure 2: Time to item acquisition (left), fraction of nodes entering an anchor zone getting an item (right)

7. A NOTE ON PERFORMANCE

In the past [11,15,21], we have conducted extensive performance evaluations of floating content-like systems, and do not re-iterate the findings here. One unexplored aspect so far has been investigating how well nodes can interact within an anchor zone using floating content (our prior work focused on how well items would float). This includes the acquisition time of new items upon entering an anchor zone and how many nodes passing through the area obtain an item.

For an initial evaluation, we choose the medium size scenario, $M50(500m, 500m)$ described in [21]: 252 nodes with 50 m radio range moving as pedestrian (160), cars (80), and trams (12) through a 4.5×3.4 km area of downtown Helsinki between predefined points of interests. Nodes use 2 Mbit/s net data rate, have 50 MB buffer, and post items of 100 KB–1 MB size once per hour with an anchor zone size of $r = a = 500$m, $T = 1$h, prioritize items inversely to their total resource consumption (size $\times T \times r$) for replication (as defined by the STF scheme in [21]), and delete items outside the anchor zone only when they encounter another node. More than 90% of the items float on average for their lifetime in this scenario. About 60% of all nodes entering an anchor zone receive a copy of an item before they leave again. The 40% of missed items are largely due to nodes within an anchor zone shortly after item creation so that there was no time for the item to reach those.

Figure 2 (left) shows, the CDF of the time to acquire a item and the time spent in the anchor zone. Most nodes receive the item within a few minutes of entering the anchor zone, while the sojourn time is usually much longer. This provides room for (repeated) item-based interactions. Figure 2 (right), depicts the CCDF of the nodes getting an item, depending on sojourn time. The X axis shows the fraction of data items acquired out of the total number of items a node could get. For longer sojourn times, on the order of a few minutes, most nodes receive most items from the area, but nodes staying only a short time may miss a large number of items. This suggests that environments in which nodes don't just move in and out quickly (such as flea markets, squares, bars, etc.) should offer good coverage to most nodes.

Contrasting these results to our application scenarios in Section 6, acquisition times on the order of a few minutes seem appropriate for them. Naturally, for chat-like applications, this may be on the long side, but in general, all the example applications seem feasible with these kinds of acquisition times. Obviously, this may affect future applications, and anyone designing applications for floating content, should design them around these constraints.

For application-specific evaluation, we use as example the local auction: We use the same setup as above and place five sellers as stationary nodes in different popular spots in Helsinki. They generate 1–4 offers per hour with $T_{ttl} = 1h$ with anchor zone sizes of 200m and 500m. An offer is 100 KB–1 MB in size (e.g., includes an image). Pedestrians may bid on the offer (cars and trams will assist in floating but won't place bids): they bid with a 5% probability when receiving the offer or another bid. Each bidder increases the bid she receives by one unit and includes the complete original message, growing its size by 1 KB. We run simulations with ten different random seeds for 12 hours and calculate the mean of how many bids are placed and the fraction received (rcvd) by the seller and how close the value of the bid gets to the maximum offered (value). Table 2 summarizes the results for 4 offers per hour; the performance is similar with smaller loads, except that more bids are generated for the 500m anchor zones, an indication of congestion.

Node	Bids	rcvd	value	bids	rcvd	value
1	5.3	.73	.96	39	.70	.92
2	10.1	.61	.88	142	.53	.88
3	3.1	.56	.89	67	.53	.83
4	11.8	.76	.94	149	.58	.90
5	0.5	.14	.50	6	.10	.55

Table 2: Sample auction application performance

We find that the floating concept approach works reasonably well (unless the node picks a too little frequented position as node 5): the majority of bids are received and they get close to the maximum bid sent. Since this is not eBay but rather a way for a seller to reach out to more potential buyers, probabilistic operation fulfills its task. Also, a smaller anchor zone size appears preferable: bidders are not so far away so that their bids are more likely to reach the seller; since the sellers stationary, their messages will float even with such a small anchor zone size.

8. SECURITY

We deliberately exclude security features because different types of applications might have widely differing requirements, ranging from no security features at all, to authenticated or encrypted items. Therefore, we leave security up to the applications and outside our API and basic system model. An application can use the metadata β to include digital signatures or information (algorithm, key ref) about how an item B is encrypted. Key management is also beyond the scope of the API. In a similar vein, we leave lower level security to the protocol loop, putting minimal requirements on the underlying "network". If a protocol loop (or link layer) supports security this can be leveraged, but sensitive applications would not rely on hop-by-hop security in the first place.

What seems contradictory to the trends in infrastructure networks, where security should be integral, appears quite suitable in our case because the location dependency of posting and responding to items has two nice properties: 1) The requirement to be inside an anchor zone when posting an item, which is validated by all entities asked to replicate the item, limits the region a node can do damage to at any given time, especially when prioritizing items for replication inversely proportional to their total resource consumption, i.e., anchor zone size, lifetime, and data volume, as e.g., suggested in [21]. 2) Applications using the API may often post items whose content has local relevance so that its validity could be verified by a user receiving an item ("Is there really free beer?"). These may suffice for many applications while the option remains to build additional security on top.

As an example, consider the case of subsuming existing items as per section 5.3. Implementing this in a secure manner requires an application to sign all *posted* items with a private key, e.g., of a self-signed certificate. The signature and certificate are included in the item metadata β and if the originator of an item wants to subsume

it, this can easily be verified by any node by checking the signature of the new item. Even if an attacker could intercept items and replace the signature and the key, it would need to catch all copies of a replicated item and, for a label generated with a keyed hash, the *label* of the item would no longer match. The originator could even delegate the power to subsume to other devices, by including a certificate in the metadata. While this is not a means for authenticating the content, it allows nodes to verify that the subsuming was done by an authorized node.

9. CONCLUSION

We introduced a system model and minimal API for applications to take advantage of opportunistic content sharing paradigms as realized by *Floating Content* and showed how to build more sophisticated application logic on top. Our examples hint that such an API is sufficient to meet the demands of quite diverse applications. They serve as initial validation, but also give us confidence into its ability to support more complex applications. Our evaluation results show the general feasibility of floating content applications and demonstrate that our API is able to meet their demands.

Acknowledgments

This work was supported by the Academy of Finland in the RESMAN project (grant no. 134363), and by TEKES as part of the Future Internet program of TIVIT (Finnish Strategic Centre for Science, Technology and Innovation in the field of ICT).

10. REFERENCES

[1] S. Amini, J. Lindqvist, J. Hong, J. Lin, E. Toch, and N. Sadeh. Caché: Caching Location-Enhanced Content to Improve User Privacy. In *Proc. ACM MobiSys*, 2011.

[2] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[3] S. Carter, E. Churchill, L. Denoue, and J. Helfman. Digital graffiti: public annotation of multimedia content. In *Conference on Human Factors in Computing Systems*, pages 1207–1210, 2004.

[4] D. D. Clark and D. L. Tennenhouse. Architectural Considerations for a new Generation of Protocols. In *Proceedings of ACM SIGCOMM*, 1990.

[5] M. Demmer, K. Fall, T. Koponen, and S. Shenker. Towards a Modern Communications API. In *Proc. ACM HotNets*, 2007.

[6] S. Floyd, V. Jacobson, S. McCanne, C.-G. Liu, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. In *Proc. ACM SIGCOMM*, 1995.

[7] M. Handley and J. Crowcroft. Network Text Editor (NTE): A scalable shared text editor for the Mbone. In *Proc. ACM SIGCOMM*, 1997.

[8] O. R. Helgason, E. A. Yavuz, S. T. Kouyoumdjieva, L. Pajevic, and G. Karlsson. A Mobile Peer-to-Peer System for Opportunistic Content-Centric Networking. In *Proc. ACM MobiHeld*, 2010.

[9] P. Hui, J. Crowcroft, and E. Yoneki. Bubble rap: Social-based forwarding in delay tolerant networks. In *Proc. of ACM MobiHoc*, 2008.

[10] P. Hui, J. Leguay, J. Crowcroft, J. Scott, T. Friedman, and V. Conan. Osmosis in Pocket Switched Networks. In *ChinaCom*, 2006.

[11] E. Hyytiä, J. Virtamo, P. Lassila, J. Kangasharju, and J. Ott. When does content float? characterizing availability of anchored information in opportunistic content sharing. In *IEEE Infocom*, Shanghai, China, Apr. 2011.

[12] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard. Networking Named Content. In *Proc. ACM CoNEXT*, 2009.

[13] S. Jaiswal and A. Nandi. Trust No One: A Decentralized Matching Service for Privacy in Location Based Services. In *Proc. ACM MobiHeld workshop*, 2010.

[14] P. Jokela, A. Zahemszky, C. E. Rothenberg, S. Arianfar, and P. Nikander. LIPSIN: Line Speed Publish/Subscribe Inter-Networking. In *Proc. ACM SIGCOMM*, 2009.

[15] J. Kangasharju, J. Ott, and O. Karkulahti. Floating Content: Information Availability in Urban Environments. In *Proc. of IEEE Percom 2010, Work in Progress session*, March 2010.

[16] T. Koponen, M. Chawla, B.-G. Chun, A. Ermolinskiy, K. H. Kim, S. Shenker, and I. Stoica. A data-oriented (and beyond) network architecture. In *Proc. ACM SIGCOMM*, 2007.

[17] E. Koukoumidis, L.-S. Peh, and M. Martonosi. RegReS: Adaptively Maintaining a Target Density of Regional Services in Opportunistic Vehicular Networks. In *Proc. IEEE PerCom*, 2011.

[18] V. Lenders, M. May, G. Karlsson, and C. Wacha. Wireless ad hoc podcasting. *ACM/SIGMOBILE Mobile Comp. and Comm. Rev.*, 12(1), 2008.

[19] B. Liu, B. Khorashadi, D. Ghosal, C.-N. Chuah, and M. Zhang. Assessing the VANET's local information storage capability under different traffic mobility. In *Proc. IEEE Infocom*, 2010.

[20] D. E. Murray, E. Yoneki, J. Crowcroft, and S. Hand. The Case for Crowd Computing. In *Proc. of MobiHeld*, 2010.

[21] J. Ott, E. Hyytiä, P. Lassila, T. Vaegs, and J. Kangasharju. Floating Content: Information Sharing in Urban Areas. In *IEEE Percom*, 2011.

[22] A. Passarella, M. Kumar, M. Conti, and E. Borgia. Minimum-Delay Service Provisioning in Opportunistic Networks. *IEEE Transactions on Parallel and Distributed Systems*, August 2011.

[23] M. Pitkänen, T. Kärkkäinen, J. Greifenberg, and J. Ott. Searching for Content in Mobile DTNs. In *Proc. of IEEE PerCom*, 2009.

[24] N. Thompson, R. Crepaldi, and R. Kravets. Locus: A location-based data overlay for disruption- tolerant networks. In *Proc. ACM CHANTS*, 2010.

[25] A. Villalba Castro, G. Di Marzo Serugendo, and D. Konstantas. Hovering information: Self- organizing information that finds its own storage. Tech. Rep. BBKCS707, Birkbeck College, 2007.

[26] E. Yoneki, I. Baltopoulos, and J. Crowcroft. D^3N: Programming Distributed Computation in Pocket Switched Networks. In *Proc. ACM MobiHeld*, 2009.