CHITCHAT: Navigating Tradeoffs in Device-to-Device Context Sharing

Sungmin Cho and Christine Julien
The Center for Advanced Research in Software Engineering
The University of Texas at Austin
Email: {smcho, c.julien}@utexas.edu

***Abstract*—Acquiring local context information and sharing it among co-located devices is critical for emerging pervasive computing applications. The devices belonging to a group of co-located people may need to detect a shared activity (e.g., a meeting) to adapt their devices to support the activity. Today's devices are almost universally equipped with device-to-device communication that easily enables direct context sharing. While existing context sharing models tend not to consider devices' resource limitations or users' constraints, enabling devices to directly share context has significant benefits for efficiency, cost, and privacy. However, as we demonstrate quantitatively, when devices share context via device-to-device communication, it needs to be represented in a size-efficient way that does not sacrifice its expressiveness or accuracy. We present CHITCHAT, a suite of context representations that allows application developers to tune tradeoffs between the size of the representation, the flexibility of the application to update context information, the energy required to create and share context, and the quality of the information shared. We can substantially reduce the size of context representation (thereby reducing applications' overheads when they share their contexts with one another) with only a minimal reduction in the quality of shared contexts.**

I. INTRODUCTION

Emerging pervasive computing applications need expressive representations of context that they can share directly with one another even given limited computation, storage, communication, and energy. In this paper, we enable devices to share views of their context and subsequently generate an aggregate view of the space and time in which they are located. The resulting notions of context enable new application behaviors. For example, if a set of devices share low-level context collected in the same area of a building, they may ascertain that the devices belong to users participating in a meeting. An application could then adapt the devices to support the meeting, for example to cooperatively select the largest screen to be a shared display or a centrally located device to act as a microphone to capture audio. This paper focuses not on the new application behavior but on the enabling capabilities of representing and sharing transient and personal context.

Modern mobile devices are capable of not only infrastructure connections but also device-to-device (D2D) connectivity, e.g., via WiFi-Direct or Bluetooth Low Energy (BLE). These facilities enable co-located devices to directly coordinate. Commodity devices and embedded sensors can also collect myriad information about us and our instantaneous situations, enabling the long vision of context-awareness [1]. We present CHITCHAT, which combines the increasing communication and context sensing capabilities of our devices to support creating and sharing views of local context. In CHITCHAT, we take a straightforward but expressive view of context to include any information that characterizes an entity's situation [11], where the entity can be a person, location, or any object relevant to the interaction between a user and application. We capture a *context* as a set of attributes, each as a (*label*, *value*) pair in a *context summary*. We refer to the set of labels as the context *schema*. We aim to create context summaries that are succinct yet able to accurately describe any situation of interest to a pervasive computing application.

Consider an emergency in which communication infrastructures are inoperable. Networks of D2D connections can help rescue workers exchange context characterizing the state of the structural environment, the presence and conditions of casualties, etc. A view of this context shared by rescuers could be used to create a response plan or stage and deploy resources. In an open-air market, visitors' devices could share context about their locations, shopping habits, or other aspects of their state or mindset. A network created from the visitors' devices could exchange context among nearby visitors, enabling applications to, for example, hint vintage lovers to visit a certain market. These are highly personalized applications that require, underneath, some view of a shared local context. The context is transient, time- and location-sensitive, and relevant primarily to others that share the same space and time. Key to enabling these applications is to provide an efficient mechanism for sharing context over D2D links.

Existing context representations [8], [15] are mostly suited to the web; they are very flexible (i.e., it is easy to update the schema and the values), expressive, and have values of high quality. As a result, the representations are quite large, making them not well suited to D2D exchange by devices with limited energy and communication bandwidth [5]. We are therefore motivated to create much smaller context representations for lightweight interactive personal sharing [21]. We must also consider the tradeoffs that come with a size reduction. For instance, it is possible to establish a schema *a priori* and share only values; this can drastically reduce communication overhead, but it also radically limits the ability for applications to change the context mechanism. Likewise, a complex representation that achieves a significant size reduction [14] may be expensive to compute; applications must consider the

power consumed in computing a smaller context summary against the power saved from reduced communication. Finally, any degradation in data quality that results from reducing the representation's size must be tolerable by applications.

CHITCHAT is a suite of context representations that allows applications to tradeoff the representation's size, quality, flexibility to be updated, and energy required to be created and shared. CHITCHAT introduces new probabilistic data structures based on a Bloomier filter [7]. In our previous work, we showed how to use a Bloomier filter to store the (*label*, *value*) pairs constituting context attributes [14]. When an application queries the structure with a specific label, if that attribute was inserted, the structure returns the correct value. Otherwise, it returns an empty value with high probability. However, it is possible that an incorrect value is returned for an attribute that was never inserted, i.e., a *false positive* occurs. Obviously, a key goal is to minimize false positives. CHITCHAT introduces two Bloomier filter based structures to further reduce a context summary's size and add the ability to update the structure on-the-fly. First is the *folded Bloomier filter*, which takes advantage of variable data widths to reduce the size of the structure without impacting the false positive rate. Second is the *complete Bloomier filter*, which maintains many of the size reductions, guarantees a zero false positive rate under certain conditions, and adds the ability to update context values in an already constructed summary. These contributions are coupled with techniques that rely on context semantics to recover correct context information. Our contributions are:

- We create concrete a set of CHITCHAT context types (Section III) that aid in achieving size efficiency.
- We introduce our Bloomier filter extensions (Section IV) and show how they can be made highly size efficient and updatable while retaining high quality information.
- We introduce techniques to recover correct values using *semantic* content of a context summary. This allows us to reduce the context summary size even more without sacrificing quality. We have integrated these techniques into a context processing engine [9]; here we show how they can help reduce the size of context representation.
- We show that applications can select among our structures to tradeoff the structure's size, power consumed, and communication overhead (Sections V and VI).

II. RELATED WORK

Context and context-aware computing have been extensively surveyed [11], as have context representations [23]. In our own prior work, we expressed context as a combination of local and shared information [19], provided a basic framework for sharing succinct context information in a pervasive computing network [14], and used context to express emergent properties of groups [18]. In this paper we use the same basic structure but substantially enhance both the data structure and algorithms to enable significant space gains without degrading the quality of the context representation. Our notion of quality is based on the accuracy of values represented and on established notions of *expressiveness*, in terms of the ability of competing representations to effectively express the same thing.

Context information is often represented using an *ontology* [2], [22], a formal description of concepts in a domain of discourse (classes), with properties of each class, and restrictions on those properties. Ontologies have been used for social context representation and automated context reasoning in a multi-agent system [3]. In CHITCHAT, we avoid using an ontology so that we can reduce the amount of knowledge that coordinating parties have to share *a priori*. However, we do adopt some properties inspired by ontology-based approaches to define constraints on relationships between attributes of contexts. Representing, storing, and sharing context are often supported in the web using XML and JSON (JavaScript Object Notation) [24]. These representations are suitable for applications in situations where resource constraints (in terms of communication and energy) are not of significant concern.

Pervasive computing devices' abilities to communicate directly with one another have increased dramatically. Initially, such capabilities were used to bootstrap discovery of nearby resources, before shifting interactions to infrastructure-supported communication [10]. More recently, D2D links have been used to carry increasing amounts of content, whether to augment or extend infrastructure [13], [16] or to enable direct application interaction [25]. The high-fidelity context that todays' devices can collect has the potential to reveal particularly personal information about the devices' users, so constraining sharing to a highly-localized region can better support users' privacy [17]; we have shown that D2D sharing of information can aid in preserving users' privacy [27]. Given these observations, we hypothesize that the time is right to push context sharing into this domain as well.

A significant remaining hurdle relates to resource constraints of devices (in energy, storage, and communication) and links (in latency and bandwidth). Existing context representations are large, and sharing them over D2D links is prohibitively expensive. A simple context description from a device in an emergency situation required 183 bytes to represent in JSON, which, using CHITCHAT's techniques, can be reduced to 33 bytes without losing expressiveness; further, we can also reduce by more than 32% the energy required to share this information using a state-of-the-art D2D link.

These reasons motivate trading the size of a context representation for the energy required to create and share it and the quality of its information. Our hypothesis is that we can take advantage of flexibility in the quality of a context representation to dramatically reduce its size. In CHITCHAT, we use a Bloomier filter as the basic data structure to represent context. A Bloom filter [4] is a size efficient data structure to capture set membership. A Bloomier filter [6], [7] associates a value with each set member. Bloomier filters have been used for purposes similar to ours to ascertain approximate group membership [12]. In CHITCHAT, we exploit associations among context attributes and values in a context summary to reduce the size of the representation without appreciably sacrificing the quality of the stored information.

III. CHITCHAT CONTEXT SUMMARIES

A *context summary*'s attributes describe an entity. We provide a set of scenarios in which context plays a pivotal role and define data types common to context values. We then introduce CHITCHAT's *filters*, which allow a context summary's contents (and the recipient's own situation) to determine whether values in the summary are likely false positives. Our prior work [9] provided an API for applications to specify these filters. In this paper, we apply the filters to our new context structures to help preserve the semantic value of the information they contain. These filters allow us to use structures that have theoretically high false positive rates that can be reduced in practice.

A. Scenarios

One category of scenarios for which D2D sharing is particularly applicable is when communication infrastructures are unusable. The goal is to collect and share context about individuals and the environment; the information could be critical in analyzing conditions to form an evacuation plan.
An earthquake has caused local blackouts and dangerous situations, trapping people and leaving the communication infrastructures inoperable. An individual's context is:

```
{ "latitude": [30, 25, 38, 2],
  "longitude": [-17, 47, 11, 0],
  "time": [11, 21],
  "date":[2015, 10, 11],
  "age": 23,
  "name": "Brian Taylor",
  "temperature": 41,
  "temperature unit": "C",
  "message": "Trapped in room 121" }
```

Similarly, information from structural sensors that survived the earthquake can share information about the building:

```
{ "latitude": [30, 25, 38, 5],
  "longitude": [-17, 47, 11, 0],
  "time": [11, 21],
  "date":[2015, 10, 11],
  "device id": 11,
  "number of sensor": 3,
  "sensor 1 name": "temperature",
  "sensor 1 value": 28,
  "sensor 1 unit": "C",
  "sensor 2 name": "humidity",
  "sensor 2 value": 43,
  "sensor 2 unit": "%",
  "sensor 3 name": "light",
  "sensor 3 value": 121,
  "sensor 3 unit": "lux" }
```

Alternatively, a device could process the data minimally, e.g., the three `sensor 1` attributes could be replaced by two:

```
...
"temperature": 28,
"temperature unit": "C"
...
```

In our evaluation, we use the first summary because it is indicative of the summary needed when no processing is done by the microcontroller collecting the sensor value.

Another motivating scenario is hyper-localized search, with the goal of sharing information about the environment and individuals' interests to help direct the individuals.

A book lover visits an open-air book fair. Her device notes and shares her interest in modern art books. A dynamic social network formed from booksellers and other visitors with similar interests can provide a small network from which to get hints of the location of the best deals.

```
{ "latitude": [31, 25, 38, 2],
  "longitude": [-17, 42, 11, 0],
  "date":[2015, 10, 09],
  "time": [10, 21],
  "leave time": [12, 21],
  "gchat id": "mary.zwky",
  "interest category 1": "art books",
  "interest item 1": "20 century European painting",
  "interest category 2": "paper",
  "interest item 2": "Roylco R15286 Antique Paper",
  "special interest item": "Hand made notebooks" }
```

The visitor's summary could be propagated until it reaches an art bookseller who could follow up via gchat. Alternatively, the context could be compared to that of similar visitors, bootstrapping sharing other context, e.g., sellers' service quality.

CHITCHAT applies generally to situations that involve contexts that exhibit the following characteristics:

- Context information is hyper-localized in space and time.
- Context attributes are related to each other.
- Strings are frequently used to describe context.
- Many context values need not be extremely accurate.
- Context values often have range limits.

We next examine these types more closely, looking at representations best suited to compact context summaries.

B. CHITCHAT *Types*

CHITCHAT defines data types tailored to pervasive computing context; specific examples are shown in Table I. We support multiple integral types and assume only single precision (32 bit) floating point numbers (a higher precision is not commonly required for context). We use Pascal-style strings, with the length as the first element. We also define special types to aid in efficiently packing data, e.g., a "level" type that scales from 1 and 10. Dates include a year (7 bits), month (4 bits), and day (5 bits) and times have hours (5 bits) and minutes (6 bits)[1]. Both latitude and longitude are expressed in degrees (± 90 for latitude and ± 180 for longitude), minutes, seconds, and subseconds. This gives sufficient precision ($\sim \pm 30cm$), given that the precision of a typical GPS unit is $\sim 1 - 10m$.

TABLE I: Example data types for contexts

Type	Bits	Bytes	Range	Encoding
Boolean	1	1	(0,1)	
Unsigned byte	8	1	(0, 255)	
Byte	8	1	(-128, 127)	
Float	32	4		IEEE 754
String	$n \times 8$	n		Pascal
Age	7	1	(0, 127)	
Level	4	1	(1, 10)	
Date	16	2		(7,4,5)
Time	11	2		(5,6)
Latitude	27	4		(8,6,6,7)
Longitude	28	4		(9,6,6,7)

[1]Applications requiring second granularity can easily add another integer value; we simply do not support seconds in the default time type.

C. False Positive Filters

A *false positive* occurs when querying a context summary returns a "junk" value for an attribute that was not inserted. To mitigate false positives, CHITCHAT developers use filters to constrain reasonable values based on application semantics. For instance, if an application queries the first of our example summaries for the attribute "building id," the summary could (in rare instances) return a junk value. An application could identify this as a false positive if it is not a real building number or if it does not make sense given the location.

Applications specify constraints on acceptable values at a variety of levels of abstraction. An *innate* filter identifies instances in which a value is not reasonable for its attribute. In our book market, a stated "leave time" of `[21, 10]` is likely a false positive because it is more than an hour after closing time. A *correlated* filter states that two (or more) attributes must be found together. For example, we might require that a summary with a longitude also has a latitude; a recommendation application may require that, if a bookseller is listed, there must be a valid rating for the bookseller. Formally, $\mathcal{C}(a_1, \ldots a_i) \implies a'$ indicates that, if a summary has valid values for attributes $a_1 \ldots a_i$, then it must have a valid value for a'. Finally, a *situational* filter, $\mathcal{S}$, checks a set of attributes $\{a_i, \ldots, a_j\}$, considering both the values in the context summary and the application's current situation: $\{a_1, \ldots, a_j\} \models \mathcal{S}(f(a_1), \ldots, f(a_j), context)$. A shopper should expect a bookseller to be nearby, i.e., that the location in their context summary is within a specified distance. Table II shows some example $\mathcal{S}$ filters.

TABLE II: Example situational relations

Context Attributes, **Situation**	$\mathcal{S}(attribute\ values, situation)$
location, **my location**	|**my location** - *location*| $\leq$ 5 km
age of kid, **elementary school**	5 $\leq$ *age of kid* $\leq$ 12
date and *time*, **today**	(*date*/*time* - **today**) $\geq$ 0

When applications specify these filters, we can shrink context summaries, increasing the number of false positives, because the filters can later remove them. Section V-B describes how CHITCHAT uses these filters to detect false positives. First, we introduce our novel structures for capturing and sharing hyper-localized context in a space-efficient way.

IV. BLOOMIER FILTERS AS CONTEXT SUMMARIES

One of our major contributions is a suite of context summary structures based on the Bloomier filter. In this section, we describe the basic Bloomier filter and an optimized version on which our work is based. We then describe the *Folded Bloomier Filter* (FBF) and *Complete Bloomier Filter* (CBF).

Basic Bloomier Filter. CHITCHAT's context summaries are built on the Bloomier filter, which is derived from the Bloom filter. A Bloom filter [4] succinctly represents set membership using a bit array m and k hash functions. To add an element, we use the k hash functions to get k positions in m and set each to 1. To test whether an element e is in the set, we check the positions associated with e's k hash values. If any position is not 1, e is not in the set. Otherwise, e is in the set *with high probability*. False positives occur if inserting other elements happens to set all k positions associated with e's hash values. A Bloomier filter [6], [7] uses a function $f(x)$ to map the set members to values. If an element e is in the input set S, the Bloomier filter's $f(e)$ should be the value associated with e. If e is not in the Bloomier filter, $f(e) = \perp$ *with high probability*.

CHITCHAT provides three structures that tradeoff complexity, size, and false positive rates. The optimized Bloomier filter (OBF), implements the original Bloomier filter [7]. Our folded Bloomier filter (FBF) targets contexts that hold values of widely varying widths, while our complete Bloomier filter (CBF) uses situations in which the context schema is (partially or fully) known to completely remove false positives for attributes in the known schema, at the cost of a small increase in the summary's size. We use an attribute's *label* as the key (e) and use $f(e)$ to refer to the associated value.

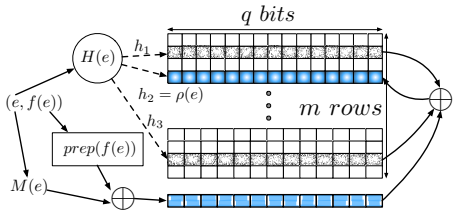
Optimized Bloomier Filter. Conceptually, an OBF is a table of size m by q. Here, m is the same as in the classical Bloom filter; $m \geq n$, where n is the number of stored associations (e, $f(e)$), and q is the maximum width of any associated value. Practically, we need only use $(q \times n) + m$ bits to store an OBF because we can simply send the n stored values and indicate, using an additional bit vector of size m, which table rows are occupied.

We define a *storage row of element* e, $\rho(e)$, to be the row where $f(e)$ is stored ($\forall e, 0 \leq \rho(e) < m$). Each $\rho(e)$ should be assigned to only one e: $\forall e_i, e_{j \neq i} \in S, \rho(e_i) \neq \rho(e_j)$. The hash function $H(e)$ generates an ordered list of k hash values for e. These hash values are indices into the table, as depicted in Fig. 1(a). A *singleton* for S is a table row that only a single element $e \in S$ hashes to. A particular $e \in S$ may have multiple singletons; for each e we select the singleton with the smallest index to be $\rho(e)$. The original presentation of the OBF [7] presents an algorithm for finding the $\rho(e)$ for all keys in a set S and determining the order of insertion of the elements in S to ensure the correctness of the OBF.

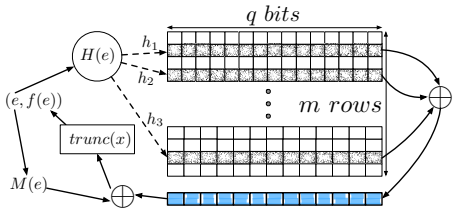
Fig. 1(a) shows table creation. Attribute e is used to generate k hash values and a mask $M(e)$. The mask is used to increase the randomness in data inserted in the table. The function $prep(f(e))$ prepends 0s to $f(e)$ to make q bits. This is XORed with $M(e)$ and the contents of the table at the locations indicated by the $k-1$ hash values that are not e's $\rho(e)$.

It may seem that we could simply store the associated value directly in $\rho(e)$. However, the receiver does not know which keys were inserted. The process of retrieving a value from an OBF is shown in Fig. 1(b). Because we do not know S, we cannot compute $\rho(e)$. Instead, we retrieve the values stored at all k locations from $H(e)$ and XOR them together with the same mask to retrieve e's q-bit value. We use $trunc(x)$ to remove any prepended 0's and retrieve $f(e)$.

For $e' \notin S$, if all k rows from $H(e')$ are empty, we can confidently return $f(e') = \perp$. It is more likely that one or more of the k rows contains some value. Given that e' was not inserted, the returned $f(e')$ is just random data. The use of the mask upon inserting data increases the innate randomness, which increases the likelihood that the data will not be well-formed



(a) Creation



(b) Retrieval

Fig. 1: OBF Table Creation Process

for the type expected. We use the application-specified filters introduced in Section III-C to catch these misrepresentations; we describe this process more in Section V-B.

Folded Bloomier Filter. The OBF has two critical limitations. First, it requires a string's length be less than the table's width (q) or the application has to handle segmenting the string into components that are each smaller than q. Second, the OBF is not size efficient when the stored types' sizes are diverse since the table width must accommodate the largest type.

Our folded Bloomier Filter (FBF) folds a large width value into multiple smaller width values. Fig. 2 shows an OBF summary with three values: {`(name, Jim)`, `(age, 7)`, `(time, (12,00))`}. The time is encoded into two bytes (0x03, 0x00), and, as a Pascal-style string, "Jim" becomes (3, 'J', 'i', 'm'). The grayed bytes show prepended zeroes[2].

Fig. 2: An OBF

Fig. 3 shows the same values folded into an FBF. The FBF is an optimization driven by properties of context structures. The FBF uses an OBF table at its core but pre-processes input associations $(e, f(e))$. For example, an eight-byte string {`(name, humidity)`} could become {`(name0, humi)`, `(name1, dity)`} before being stored in an OBF with a width of four bytes. Likewise, the FBF post-processes retrieved data to get $f(e)$; multiple values are retrieved from the table and combined. We augment the labels with a sequence number to enable the pieces of a value to be reassembled in order.

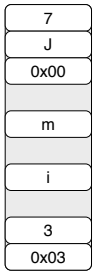


Fig. 3: An FBF

We assume that the table width is an integer multiple of bytes, both for simplicity and because not doing so entails added processing overhead. A table width of one byte is theoretically optimal, but this forces m to be large and offsets the size benefits gained from folding. Folding a table causes an increase in the false positive rate, but we are able to take advantage of our application-provided filters to effectively detect and eliminate these false positives.

Complete Bloomier Filter. The greedy algorithm to find $\rho(e)$ for each key requires discovering and enforcing an insertion order [7]. The structure can be made orderless when every e can be associated with a singleton $\rho(e)$ on the first search, meaning that each value can be directly stored in (and retrieved from) the table without masking and XORing. In these cases, the table has the property that, for all e of the k rows from $H(e)$, only the row $\rho(e)$ is not empty. The potential disadvantage is that the number of rows in the table must be made large enough to ensure a unique position for each key.

Using this property, we can make a false positive free structure. Imagine that our second application desires to ensure absolutely zero false positives for a subset of the attributes, S' = {`special interest item`, `gchat id`}. The motivation may vary; for the `special interest item` users may simply not tolerate incorrect values). On the other hand, because `gchat id` can be any string, filtering false positives can be very difficult, so eliminating them entirely may be preferable. Recall that S is the set of attributes included in the summary. S' may or may not be a subset of S, that is, the application may choose not to insert some elements of S'. For any $e' \in S'$, the summary is guaranteed to have no false positives. For any other element, the same properties hold as for the FBF.

We construct the table to reserve a singleton $\rho(e)$ for all $e \in S \cap S'$. For any $e' \in S'$, any retrieved $f(e')$ is a true positive. Achieving this property requires a table with a larger m, and knowledge about which attributes are in S' must be shared among the participants. We call this structure a complete Bloomier Filter (CBF). The CBF carries another very important quality: the value $f(e')$ associated with $e' \in S'$ in a CBF can be updated on-the-fly without having to recreate the entire CBF. The same is not true for the OBF and the FBF.

V. Recovering Context in Chitchat

In this section we review our context summary structures then detail how Chitchat uses semantic information to drastically reduce the false positives encountered in recovering attributes from a summary. Finally, we analyze the tradeoffs of size efficiency, flexibility, energy efficiency, expressiveness, and quality (in terms of false positives) of the structures.

A. Chitchat *Context Summary Structures*

Chitchat provides seven structures to store and share context in device-to-device networks. These options make tradeoffs to allow Chitchat to adapt to different network environments. The seven options are built from five basic structures: (1) a *JSON context summary*, which represents the current state-of-the-art; (2) a *labeled context summary*, which is a simple flat dictionary that uses Chitchat types; (3) a *complete context summary*, which is very efficient but assumes the context schema is shared in advance (4) our

[2]The OBF in the figure is simplified, as it shows inserted data that has not been XORed; the space savings are the same in the unsimplified structure.

TABLE III: CHITCHAT Context Summaries

Summary	Compressible	Description	Benefits	Size
JSON	Yes	stores attributes in text-based representation; values are stored as strings; effectively a flat dictionary that uses e to look up $f(e)$	high quality; can update schema & values	large and variable
Labeled	Yes	represents attribute labels as strings associated with CHITCHAT typed values; can be viewed as flat dictionary	high quality; can update schema & values	$\sum_i^n (\|e_i\| + \|f(e_i)\|)$
Complete	No	uses attribute label e as an index into an array of $f(e)$ values; requires sharing context scheme *a priori*	high quality; very small	$\lceil \ln n \rceil \times n + \sum_i^n \|f(e_i)\|$
FBF	No	uses Bloomier filter to store context attributes; relies on semantic filters to remove false positives	very small	$n \times r + m$
CBF	No	uses Bloomier filter to store context and semantic filters to remove false positives; requires larger table (i.e., m) than FBF	small; can update values	$n \times r + m$

Folded Bloomier filter (FBF); and (5) our *Complete Bloomier filter* (CBF). CHITCHAT also provides a compressed version of the JSON and labeled summaries using lossless compression; compressing the other structures does not reduce their size.

Table III summarizes the properties and benefits of CHITCHAT's context summary structures. In the sizes, n is the number of context attributes stored, m is the size of the table underlying a Bloomier filter, and $|x|$ is the size of x's type. The Bloomier filter based summaries do not inherently have a high quality; instead they rely on CHITCHAT's semantic filters to substantially increase the quality of representation.

B. False Positive Detection Process

The values returned from CHITCHAT's context summaries may contain false positives. Our previous work [9] presented programming interfaces by which developers provide application-specific definitions of filters to apply to contexts. In this paper, we define how the algorithms that implement the filters from Section III-C interact with our new structures.

To motivate the need for our semantic filters, we gauge our summaries' true negative rate, or the rate at which our structures immediately return $\perp$ when an attribute has not been inserted; this occurs when all k table locations indicated by $H(e)$ are empty, which is dependent on k, m (the number of table locations), and n (the number of inserted attributes). Theoretically, it is $\prod_{i=0}^{k-1} \frac{m-n-i}{m-i}$. Empirically, we generated 100,000 summaries, each with 5 to 10 attributes of random types. We queried each summary for an attribute that was not inserted and counted the number of times the value was a true negative. Our empirical results match the theory. Further, in our experience, our summaries achieve the best size efficiency when $m/n = 1.22$; at this setting the true negative rate is $3 - 4\%$, that is $96 - 97\%$ of true negatives are undetected. Thus, the ability to remove false positives is critical.

To remove false positives, CHITCHAT applies the filters from Section III-C in succession. Before applying the application-level filters, however, we use the fact that the space available to store a value (2^q bits, in a table of width q) is often larger than the value's range. The value of $f(e')$ from $e' \notin S$ will be between 0 and $2^q - 1$. When $f(e)$'s type uses b bits, the upper $(q - b)$ bits should be zero. If they are not, we have found the simplest false positive. For remaining apparently positive results, the *innate filter* asks whether a value is valid given the innate range and type encoding. After applying the innate filter, any seemingly valid values are passed to the *correlated filter*, which uses relationships among attributes in the same summary to detect false positives. This is a powerful filtering technique, considering that the probability of randomly generating correct values is very low. Applications can easily provide sophisticated correlation relations; for example, when the contexts found together are {`location name`, `latitude`, `longitude`}, and $f(\texttt{location name}) =$ *City Park*, the location should indicate a place within *City Park*. Finally, we apply *situational filters*, which identify values in a context summary that simply do not make sense in the current situation. For instance, if I am sharing bookseller recommendations with others nearby, and a potential collaborator's context summary contains a location that indicates he is in Antarctica while I am in the United States, it can be assumed to be a false positive.

C. Experiments with False Positive Detection Process

Given this process, we measure how well CHITCHAT corrects the potentially large number of false positives that escape the basic Bloomier filter structure. We use randomly generated contexts with attributes of varying types; each value is a randomly generated bit string of the appropriate length. We query these summaries for attributes that *were not* inserted, i.e., for which CHITCHAT (after applying all of the false positive filters) should return $\perp$. The results are in Table IV.

We used the innate filters in Table I. For correlated filters, we assumed that times are found with dates and latitudes with longitudes. We also assume that every instance of age, level, temperature, or float is correlated with some string attribute (e.g., that a `sensor value` (of type float) always be accompanied by a string attribute `sensor unit` or that a `bookseller recommendation average` always appears with a string `bookseller name`). We require values of type string to be decoded into valid (i.e., printable) strings. The probability that a series of random bits can be decoded into a valid string is $1/256 \times \alpha^{n'} \times \frac{1-\alpha^{(256-n')}}{1-\alpha}$, where n' is the minimum allowable string length, and $\alpha = 95/256$ is the ratio of ASCII printable characters to non-printable ones. This results in the 0.085% theoretical false positive rate for strings when $n' = 2$, which, because we require any float to be correlated with a string, becomes the (theoretical) false positive rate for float values in the table. The theoretical values for age, level, and temperature are calculated similarly, though they are overall lower since the innate filters have first removed some false positives. For situational filters, we used:

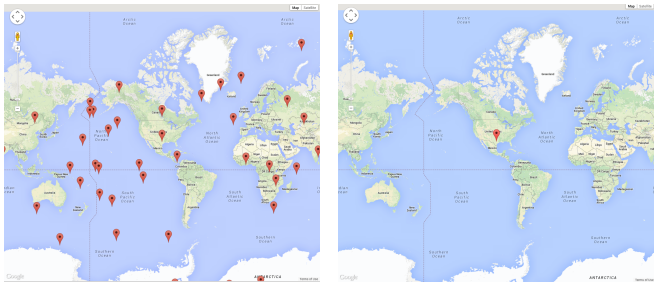
Temperature: $|temperature -$ **current temperature**$| \leq 25°C$
Latitude: $|location -$ **my location**$| \leq 10$ km
Time: $|date/time -$ **today**$| \leq 2$ months

TABLE IV: False positive (*fp*) probabilities (FBF)

Type	$fp_{innate}(\%)$		$fp_{correlate}(\%)$		$fp_{situation}(\%)$	
	theory	exp.	theory	exp.	theory	exp.
Boolean	0.39	0.38				
Age	50.0	50.3	0.0427	0.0407		
Level	4.29	4.28	0.0037	0.0034		
Float	100.0	99.9	0.085	0.082		
Temp.	43.3	43.3	0.037	0.036	0.017	0.014
Latitude	1.5	1.5	0.045	0.051	2.8×10^{-8}	0
Time	2.2	1.9	0.12	0.11	0.0042	0.001

Table IV gives the likelihood that a value returned after each filter is a false positive, e.g., only 1.5% of latitude values that pass the innate filter are not actual latitude values. On the other hand, almost 100% of the float values that pass the innate filter are false positives. In all cases, after applying all filters, the false positive probabilities are very small.

Next we highlight tradeoffs between the FBF and CBF. Consider 100,000 randomly generated summaries that do not contain a valid location attribute, and one summary that does. After applying innate and correlated filters to the FBF, we have 43 false positives and the one true positive, plotted in Fig. 4. When we place either longitude or latitude in the reserved set (S'), a CBF resolves 100% of the false positives. Fig. 4 showcases the potential of situational filters; with any requirement of nearness to a point of interest, the FBF can remove 100% of the false positives; a looser situational filter that requires a point to be on land removes 72.1% of them.



(a) True + False Positives (b) True Positive

Fig. 4

VI. Evaluations

We evaluate the size and energy efficiency of CHITCHAT's structures under various scenarios. In addition to this benchmarking, we provide a rudimentary demonstration via simulation of CHITCHAT's use for communicating context information in a pervasive computing network. Our contributions are not in the space of context distribution protocols, so this evaluation assumes a very basic epidemic protocol.

Size Efficiency. To identify tradeoffs, we evaluate and compare CHITCHAT's seven alternative structures. For compression of JSON and labeled context summaries, we use ZLIB. We first compare the sizes while varying the table width. We found these results to be insensitive to the number of hash functions; we use $k = 3$. As Fig. 5 shows, the complete context summary is the most size efficient (but it is the least flexible), and the (uncompressed) JSON context summary is the least size efficient (but the most flexible). We consider these to be upper and lower bounds on context representation sizes.

Among the other options, the FBF achieves the best size efficiency. At larger table widths, the sizes of the FBF and CBF converge because q approaches the size of the largest value. In the two context summaries from the first scenario, most attributes are small, making the FBF quite efficient at small widths. For the second scenario, numerous large strings make the total table size larger for CBFs with small widths. The benefits of the FBF over the CBF are evident; reserving space for large strings in narrow tables can reduce size efficiency.

TABLE V: Context Summary Scenarios

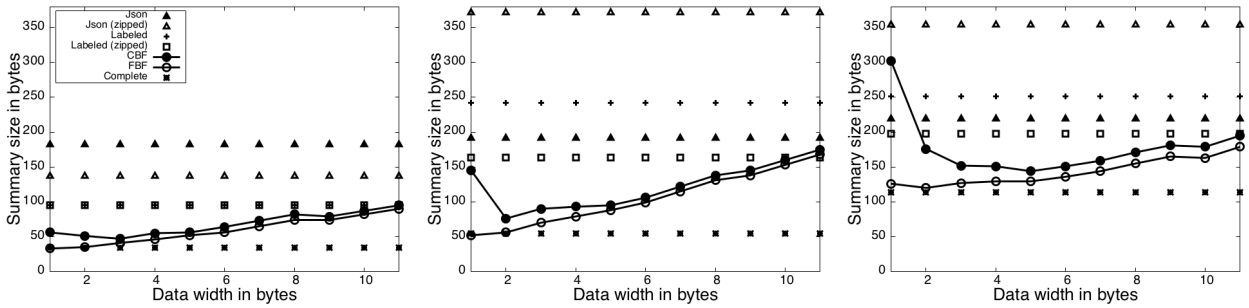
Scenario	Description	Properties
s1	sharing information about nearby soccer players to start a pickup game	short strings (less than 10 characters) and small types
s2	sharing ride recommendations in an amusement park	
s3	sharing information about nearby known friends	long strings (more than 50 characters) or large types
s4	restaurant recommendations	
s5	firefighter offering supporting services at a fire scene	multiple intermediate sized strings
s6	bus schedule information	

We created six additional summaries (shown in Table V) driven by real applications. Table VI shows, for each scenario, the reduction in size versus the (uncompressed) JSON summary and increase versus the complete summary. In the table, c1 is a context without any strings, and c2 has only strings ranging in length from 4 to 38 characters. Our structures substantially reduce the size, meeting (and in some cases exceeding) the ability of the complete context summary without having to share the context schema *a priori*. The savings with the CBF are less than with the FBF because the former reserves attributes for which it guarantees a complete absence of false positives; in these experiments, we put the entire context schema in the CBF's reserved set.

TABLE VI: Size efficiency

Scenario	Reduction (%) vs. JSON		Increase (%) vs. complete	
	FBF	CBF	FBF	CBF
s1	76.42	67.45	6.38	46.81
s2	80.47	74.22	6.38	40.43
s3	65.49	55.75	1.30	29.87
s4	63.59	51.09	3.08	38.46
s5	65.99	52.63	1.20	40.96
s6	62.40	31.82	4.60	89.66
c1	87.42	84.91	-4.76	14.29
c2	46.55	28.16	4.49	40.45

Energy Efficiency. The Trepn [26] profiler allows Android applications to measure performance and power consumption. We used Trepn with a MotoG XT 1032 (we find no noticeable differences for other devices) to measure the power consumed encoding context into summaries and sharing them over WiFi.



(a) Scenario 1: Earthquake (User context) (b) Scenario 1: Earthquake (Sensor context) (c) Scenario 2: Open Air Market

Fig. 5: Size Comparison

We invoked Trepn at its maximum sampling frequency of 10Hz. We profiled the same function for the same settings multiple times until the total time reached into minutes; we averaged the results using the number of contexts created and sent and used the device's baseline power consumption to compute only the overhead of encoding and sending contexts. Trepn provides stable and accurate power measurements for profile periods of seconds to minutes. It was more difficult to obtain accurate measurements at the timescales of WiFi communication (especially for small context summaries). We iterated over sending summaries thousands of times took an average to obtain a value for sending one summary.

We first measured power consumption for encoding contexts into FBF summaries. As Table VII shows, encoding into a smaller table requires more energy. When the table width is small, elements are packed more tightly (hence the large size reductions). However, the algorithm that finds a satisfying assignment of elements to storage rows has to search longer, resulting in more processing; for example, the processing time for c2 was 5.8s with $q = 16$ but only 0.2ms with $q = 64$. Context c1, with only fixed width elements, does not show a substantive difference in performance for different table sizes.

TABLE VII: Power Consumption measured in mW

	JSON	FBF ($q = 16$)		FBF ($q = 64$)	
	size (bytes)	Reduction in size (%)	Power	Reduction in size (%)	Power
s1	212	76.42	7.32	57.55	0.35
s2	256	80.47	6.41	58.20	0.98
s3	226	65.49	17.85	53.10	1.05
s4	184	63.59	9.07	51.09	0.33
s5	247	65.99	20.63	50.61	1.33
s6	242	62.40	16.94	49.17	1.69
c1	159	87.42	0.37	63.52	0.36
c2	174	46.55	21.31	39.08	1.25

In our next experiment, we sent c1 with $q = 16$ over a WiFi link at 100ms intervals to mimic a situation in which context information is updated and shared continuously. We measured the cost of sending a JSON summary as a reference. Compared to the size reduction of 87.42%, the overall (encoding and communication) power reduction was at least 21.05%. The reduction in overall energy is not as substantial as the reduction in space, but it is still important and meaningful.

Our next experiment targeted just the cost of *sending* and not creating contexts. The goal was to determine what savings are actually due to sending smaller context summaries. This is important especially for the CBF, which can be updated without any additional processing, i.e., the energy cost of creating the context can be amortized over many transmissions. The energy reduction of sending our summary in comparison to the JSON summary was about 32% on average, while receiving our shorter summary saved about 15% on average.

Real World Context Sharing. To assess how CHITCHAT might perform in real applications, we used the ONE Simulator [20] and a basic epidemic context dissemination protocol to mock our open air market. We used a city, modeled on Manhattan, with four open air markets. We also constrained the space to create a denser environment. We call the former "*Manhattan*" and the latter "*mini-Manhattan*." Mobile users' move according to ONE's random walk model and include 41 people interested in book markets (one is a designated device d) and 40 people who are uninterested in the markets. The users interested in markets use the summary in Section III. Others share a summary selected from s1 through s4 in Table V. Markets' summaries describe inventory, opening hours, locations, and discounts. Each device periodically transmits a block of summaries to neighboring devices. This block includes the device's own summary and summaries received from others; as such, context epidemically spreads over a dynamic multi-hop network. Each block has a size budget; we used base budgets of 1K, 2K, and 5K bytes, and each device individually varies the base budget by $\pm 10\%$. To assemble a block, a device first inserts its own summary, adds received summaries with similar interests, and fills remaining space with other most recently received summaries. A budget of 1K bytes allowed around 10-20 FBF summaries or around 4-5 JSON summaries. Devices had communication ranges of 100m and link speeds of 2Mbps[3].

We assess the potential impact of CHITCHAT's reduction in the size of context on how fast context information can spread in a multihop network. We measured the average time for the d's summary to reach booksellers or for a bookseller's summary to reach d. Table VIII shows the results under budgets of 2K and 5K bytes; with a 1K byte budget, only the FBF was successful. When the base budget is 2K bytes, the

[3]We selected 2Mbps as it is the communication speed of BLE.

TABLE VIII: Dissemination Time and Success in ONE Simulation

	1K/2K byte budget						5K byte budget					
	success (%)			time (s)			success (%)			time (s)		
	FBF	Labeled	JSON	FBF	Labeled	JSON	FBF	Labeled	JSON	FBF	Labeled	JSON
Manhattan	100%	67%	67%	1248	1820	2446	100%	100%	100%	675	583	1485
mini-Manhattan	27%	11%	16%	329	315	208	78%	53%	33%	263	364	308

labeled structure is 36.7% slower than the FBF, and the JSON structure is 97.3% slower. With a 5K byte budget, the labeled structure actually outperforms the FBF. In *mini-Manhattan*, with a small budget (1K), the success rate is low because our simple dissemination scheme shares similar contexts, making it difficult to propage something new in a short amout of time. With the 5K byte budget, the JSON and labeled structures succeeded less than the FBF, and, when they succeeded, they reached the bookseller 17.1% and 38.4% slower than the FBF, respectively. These results demonstrate that the choice of the context structure to use depends on network capabilities; when communication is limited or interactions are hyper-localized, using CHITCHAT's Bloomier-filter based structures can substantially increase context sharing opportunities.

Our contributions are not in new context dissemination protocols. This network evaluation places CHITCHAT's in a larger picture that shows the *potential* for sharing context in pervasive computing. Coupled with the benchmarks above, we showed that CHITCHAT can be a practical method for context sharing because its context summary structures enable applications to effectively tradeoff size efficiency, energy efficiency, flexibility, expressiveness, and communication overhead.

ACKNOWLEDGEMENTS

This work was funded, in part, by a Samsung GRO and the NSF, #CNS-1218232. The views and conclusions are those of the authors and not of the sponsoring agencies.

VII. CONCLUSION

Representing context information flexibly, expressively, and size efficiently is critical to pervasive computing applications that share such context using limited resources. We presented CHITCHAT: a suite of context representations that provides various structures that allow application programmers to tune tradeoffs to meet application requirements. Our novel contributions include two tailored optimizations of the Bloomier filter and CHITCHAT types tailored to pervasive computing context. We showed that drastic size reduction in context summary without sacrificing expressiveness and flexibility is possible, and the possible degradation in quality that comes can be overcome by the use of simple semantic filters. Specifically, we demonstrated up to a 87.42% reduction in the size of a context representation with a near zero false positive rate for sharing context values, and at least 21.05% energy reduction when encoding and sharing contexts compared to the JSON context summary widely used in the Internet. We also demonstrated that in a real world context sharing simulation with limited budget, using FBF context structure shows a strong potential to increase context sharing opportunities.

REFERENCES

[1] M. Baldauf, S. Dustdar, and F. Rosenberg. A survey on context-aware systems. *Int'l. J. of Ad Hoc and Ubiquitous Comp.*, 2(4):263–277, 2007.
[2] T. Berners-Lee et al. The semantic web. *Scientific American*, 284(5):28–37, 2001.
[3] G. Biamino. Modeling social contexts for pervasive computing environments. In *Proc. of PerCom Workshops*, 2011.
[4] B. H. Bloom. Space/time tradeoffs in hash coding with allowable errors. *Comm. of the ACM*, 13(7):422–426, 1970.
[5] A. Carroll and G. Heiser. An analysis of power consumption in a smartphone. In *Proc. of the USENIX Annual Technical Conf.*, 2010.
[6] D. Charles and K. Chellapilla. Bloomier filters: A second look. In *Proc. of ESA*, 2008.
[7] B. Chazelle et al. The Bloomier filter: an efficient data structure for static support lookup tables. In *Proc. of SODA*, 2004.
[8] H. Chen, T. Finin, and A. Joshi. Semantic web in the context broker architecture. In *Proc. of PerCom*, pages 277–286, Mar. 2004.
[9] S. Cho and C. Julien. The grapevine context processor: Application support for efficient context sharing. In *Proc. of MOBILESoft*, 2015.
[10] M.S. Corson et al. FlashLinQ: Enabling a mobile proximal internet. *IEEE Wireless Communications*, 20(5):110–117, Oct. 2013.
[11] A.K. Dey and G.D. Abowd. Towards a better understanding of context and context-awareness. In *Proc. of CHI Workshop on the What, Who, Where, When, and How of Context-Awareness*, 2000.
[12] M. Dietzfelbinger and R. Pagh. Succinct data structures for retrieval and approximate membership. In *Proc. of ICALP*, 2008.
[13] K. Doppler et al. Device-to-device communication as an underlay to LTE-advanced networks. *IEEE Comm. Mag.*, 47(12):42–49, Dec. 2009.
[14] C.-L. Fok, E. Grim, and C. Julien. Grapevine: Efficient situational awareness in pervasive computing environments. In *Proc. of PerCom Workshops*, pages 475–478, Mar. 2012.
[15] T. Gu, H.K. Pung, and D.Q. Zhang. A service-oriented middleware for building context-aware services. *J. of Network and Computer Applications*, 28(1):1–18, Jan. 2005.
[16] B. Han et al. Mobile data offloading through opportunistic communications and social participation. *IEEE Trans. on Mobile Computing*, 11(5):821–834, 2011.
[17] Q. Jones et al. Geographic place and community information preferences. *CSCW*, 17(2–3):137–167, 2008.
[18] C. Julien. The context of coordinating groups in dynamic mobile networks. In *Proc. of Coordination*, 2011.
[19] C. Julien, A. Petz, and E. Grim. Rethinking context for pervasive computing: Adaptive shared perspectives. In *Proc. of ISPAN*, 2012.
[20] A. Keränen, Jörg Ott, and T. Kärkkäinen. The ONE simulator for DTN protocol evaluation. In *Proc. of SimuTools*, 2009.
[21] J.-S. Lee and U. Chandra. Mobile phone-to-phone personal context sharing. In *Proc. of the* 9^{th} *Int'l. Symp. on Comm. and Info. Tech.*, pages 1034–1039, Sept. 2009.
[22] N. F. Noy and D. L. McGuinness. Ontology development 101. Technical report, Stanford Knowledge Systems Laboratory, 2008.
[23] M. Perttunen, J. Riekki, and O. Lassila. Context representation and reasoning in pervasive computing: a review. *Int'l. J. of Multimedia and Ubiquitous Comp.*, 4(4):1–9, 2009.
[24] R. Reichle et al. A comprehensive context modeling framework for pervasive computing systems. In *Proc. of DAIS*, pages 281–295, 2008.
[25] V. Srinivasan, T. Kalbarczyk, and C. Julien. Disseminate: A demonstration of device-to-device media dissemination. In *Proc. of PerCom (Demonstrations)*, 2015.
[26] Trepn Profiler. https://developer.qualcomm.com/mobile-development/increase-app-performance/trepn-profiler. Accessed: 2015-04-12.
[27] M. Xing and C. Julien. Trust-based, privacy-preserving context aggregation and sharing in mobile ubiquitous computing. In *Proc. of Mobiquitous*, Dec. 2013.