The Grapevine Context Processor: Application Support for Efficient 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—**Today's ubiquitous wireless networks enable easy access to information on the web. However, when applications need access to highly transient and hyper-localized information, web access is not as efficient as sharing directly with co-located devices. This *context* information can be easily collected by commodity mobile devices and shared via device-to-device interactions. Open challenges remain in making this exchange as efficient and effective as possible. In this paper, we propose the *Grapevine Context Processor* (GCP), which provides a framework and a set of APIs for representing and sharing context information. GCP is built on the Grapevine context sharing framework, which is underpinned by a suite of novel space-efficient context representing data structures. GCP, as a context processing framework, enables application developers to easily specify context types of interest and provide relevant semantic filters for those types; we design a set of application programming interfaces (APIs) that allow easy access to and control over the data structures.**

I. Introduction

On our persistently connected mobile devices, the Internet is our primary tool for sharing and accessing information. However, our devices can collect myriad information about our hyper-localized and highly transient situations, which vastly extends the information we can share. Storing and accessing these excessive volumes of information via the Internet may not be practical, nor may it satisfy user's actual needs. On a day with particularly bad traffic, the regular bus schedule may not answer a question about when the next bus will arrive. The fact that today's sale item is nearly sold out may be critical for those who intend to visit a store for the special offer. In such instances, accessing information via the Internet may not be as efficient as sharing hyper-localized information directly with co-located peer devices. Our premise is that this situational context [4] can be easily collected by our mobile devices and shared via device-to-device interactions; applications demand that this context be shared in near-real time to ensure that it is still relevant when the applications act on it.

In resource constrained mobile environments, reducing communication overhead is of extreme importance, given its impact on costs of data access and energy consumption. Simplifying development of mobile applications is often a confounding factor, as the time to deployment is a significant influence on technology development. We present the Grapevine Context Processor (GCP), which enables developers to build applications that share context in a very size efficient way. GCP includes facilities to generate user-defined context types and APIs for interacting with and efficiently sharing these contexts. GCP encompasses novel data structures for probabilistic and space-efficient context representation, described in our previous work on the Grapevine middleware [6].

Grapevine provides a lightweight data structure for context that may sacrifice the correctness with which it stores context values in exchange for reducing the size of the representation. These context structures are simple but expressive optimizations of the Bloomier filter [3], an augmentation of a Bloom filter that not only captures set membership but also associates a value with each set member. A context captured in this structure can be shared among devices without incurring a large communication overhead. The nature of this structure results in potential misrepresentations of the stored context information. In this paper, we build the Grapevine Context Processor (GCP), which leverages application-stated constraints on context values to control the misrepresentation probability and to reconcile inconsistent values by correlating diverse bits of context across shared context structures.

When we create a context structure, we insert a set of (*attribute*, *value*) pairs. When an application later queries the structure with a specific attribute, if the attribute was inserted, Grapevine returns the correct value. If the attribute was not inserted, the structure 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., the query returns a *false positive*. GCP provides application programming interfaces to make it easy for developers to specify how to handle these inevitable false positives. Application developers provide a set of *filters* on "true" responses for a context structure query. *Innate filters* ensure that returned values are within a reasonable range for a particular type, *correlation filters* check that relationships between the values for related attributes are consistent, and *contextual filters* check the values for attributes against the context. We achieve a large reduction in the size of the context representation for real pervasive computing application scenarios while achieving a near zero false positive rate. GCP streamlines the development effort required to efficiently share context in mobile applications, especially via device-to-device communication channels.

II. Bloomier Filters for Context Representation

Context and context aware computing have been extensively surveyed [4], as have requirements and techniques for context representation [11]. In our own prior work, we have expressed context as a combination of local and shared information [9], and our Grapevine model provides a framework for sharing context information in a pervasive computing network [6]. In this previous work, we have focused on the use of context to express emergent properties of groups [8] using a basic representation of the context based on a Bloomier filter. The work in this paper extends our prior work by providing techniques that allow content- and context-based filtering of context information, largely based on its social properties.

We use *context* to describe human users, their attributes, and their relationships with the environment. A *context summary* (or just *summary*) is a structure that lists the context elements that describe the situation of some entity (e.g., a user). The *schema* (of a summary) is the list of attributes for which values are stored in the summary. As a simple example, consider one category of scenarios for which our context sharing approach is particularly applicable: connecting groups of users. The goal is generally to share context of individuals to create a collaborating group based on the shared context state.

As people arrive in a city park to spend the day, they share information about their leisure interests and skills. By sharing information about their enthusiasm for football and their associated skill levels, an organically defined group emerges that defines two well-matched teams ready for a pick-up game.

```
sport -> football
position -> goalkeeper
skill level -> 5
gchat id -> john1988
latitude -> (30, 25, 38, 2)
longitude -> (-17, 47, 11, 0)
available date -> (2014, 9, 12)
available start time -> (12, 15)
available end time -> (15, 30)
```

The above summary describes a person in the park interested in playing football. The summary includes the user's instant message id, his current location, and his available times. Clearly, different applications may include different information; further, a single user's context summary may contain information that is ultimately consumed by many different applications. When this summary is shared with opportunistically connected mobile devices, an application can automatically generate a group of people who are near each other, interested in playing football, with sufficient overlap in available times. The application could then define two "teams" based on the group's coverage of necessary positions.

There are many possible representations for a context summary. A *labeled context summary* stores both the keys and values and is the most expressive context representation. A *complete context summary*, on the contrary, stores all of the values but associates each one with an *index* into some globally known dictionary of attributes. This results in better size efficiency, but limits the expressiveness. The Grapevine approach to context summaries splits the difference to achieve the expressiveness and flexibility of a labeled summary and the size efficiency of a complete summary.

Grapevine's context summaries are built on the Bloomier filter, which has been used for purposes similar to ours in dictionary retrieval problems to ascertain approximate group membership and theoretical bounds on the reduced size of a summary representation [5]. While a Bloom filter [1] succinctly represents set membership using a bit array m and k hash functions, a Bloomier filter [2], [3] is uses an association function $f(x)$ to capture the mapping of 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*. We have derived a variety of Bloomier filter based context summary structures, but we omit the details for brevity.

III. False Positives Detection

Queries over our Bloomier filter based context summaries can result in *false positives*, which occur when a context structure responds that a value for an attribute was stored when it was not. In these situations, the structure returns a "junk" value. To mitigate these challenges, GCP allows application developers to specify filters that constrain reasonable values based on application-level semantics. In Section IV we define the APIs that developers use to specify filters. In this section, we show how the algorithms function behind the scenes to apply application-provided filters to detect false positives.

Fig. 1 shows our false positive detection process. A query immediately returns $\perp$ (i.e., a true negative) for e when the underlying Bloomier filter returns a true negative. A *false positive* results when the filter fails to return $\perp$ for a non-member, instead returning a value $v_\perp$ pulled from the bits in the underlying table.

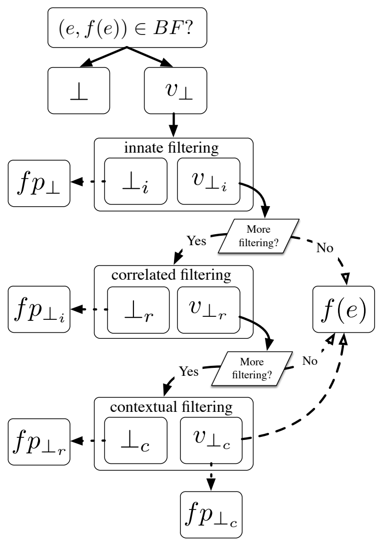

Fig. 1: False positives detection

Our first method for detecting false positives uses the property that the space available to store a value (2^q bits, given a table of width q) is often larger than the value's range. When $f(e')$ from $e' \notin S$ is retrieved, the possible value is between 0 and $2^q - 1$. When $f(e)$'s type uses b bits, the upper $(q - b)$ bits of the value should be zero; otherwise, our context summary can return $\perp$. However, a returned, apparently valid, value $(v_\perp)$ may still be a false positive with a relatively high probability. This basic filtering and the true negative detection are the first stage in Fig. 1. After applying this simple filter, any seemingly valid value $(v_\perp)$ is passed to the next stage filter, which is where our application-specified filters take over.

We refer to undetected false positives that pass the first filter as $fp_{\perp}$. Our *innate filter* detects whether a value $v_{\perp}$ is valid given the innate range and type encoding; an invalid value identified by an innate filter is labeled $\perp_i$ ($\perp_i$ is an the identified false positive; $fp_{\perp}$ are the actual false positives); as in the previous step, there may still remain undetected false positives (i.e., $fp_{\perp_i}$) that are passed to the next stage as potentially valid values ($v_{\perp_i}$).

Even when a value is innately correct, correlations between attributes, provided by application developers, may identify a value as a false positive. For instance, if we retrieve a seemingly correct longitude value but no latitude value, we can assume that the former is a false positive. Applications can easily provide sophisticated correlation relations; for example, when the contexts found together are {`location name`, `latitude`, `longitude`}, and $f($`location name`$)$ = *City Park*, the location should indicate a place within *City Park*. Invalid values detected in this way are identified as $\perp_r$.

Finally, an application's *contextual filters* detect false positives that simply do not make sense in the current situation. For instance, if the application is looking for people the user can play football with, and a player's location indicates he is in Antarctica while the user is in the United States, an application's contextual filter can indicate that is a false positive. Such violations are identified as $\perp_c$.

Filters applied earlier are easier for the programmer to define, are often reusable across applications, and are less computationally expensive. Regardless, it is possible that some false positives slip past these filters and into applications. These are the undetected false positives; we refer to them as $fp_{\perp_c}$. Our fundamental goal is to reduce the number of these false positives as much as possible while maintaining space efficiency of context structures.

We next benchmark our false positive filters' capabilities by measuring the false positive detection rates for each type of filter. We use randomly generated contexts that contain attributes of varying types; each attribute's value is a randomly generated bit string of the appropriate length. We use randomly generated context summaries to make it easy to create summaries of varying sizes. We query these context summaries for attributes that *were not* inserted, i.e., for which the application-level response (after applying all of the false positive filters) should be some form of $\perp$. The results are shown in Table I.

We used string correlations for age, level, temperature, and float (e.g., we assume correlations between {(`age of kid`, 12), (`name of kid`, John)} and {(`recommendation average`, 5.6), (`recommendation restaurant`, Sushi Palace)}). For latitude and time, we assume that latitude is not found without longitude and dates are not found without times. For contextual filters, we used:

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

In Section IV we show how the application can easily specify such a filter using the Grapevine Context Processor APIs.

The numbers in Table I give the likelihood that a value returned after each filter is a false positive, e.g., only 1.5% of the latitude values that pass the innate filter are not actual latitude values. Almost 100% of the float values that pass the innate filter are false positives. In all cases, after applying all of the filters, the false positive probabilities are very small.

TABLE I: False positives probabilities (FBF)

Type	$fp_{\perp_i}(\%)$		$fp_{\perp_r}(\%)$		$fp_{\perp_c}(\%)$	
	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

IV. Grapevine Context Processor

In this section, we present the Grapevine Context Processor (GCP) for mobile applications. GCP comprises an application specific code generator, core libraries that the generated code interacts with, and a set of simple application programming interfaces. GCP differs from our prior work [7] in that its main focus is on a size efficient way of communicating context information. CALAIS [10] supports context-aware applications with a similar goal of providing a simple language for applications to express context-dependent behaviors; CALAIS describes composite events, while GCP allows applications to tailor filters for enhancing the false positive detection rate. Furthermore, CALAIS requires connectivity to a central server.

Application developers use GCP to specify context types and false positive filters by providing a type description containing the type's properties and its correlation and contextual constraints. We use a `json` file that is processed by the GCP compiler to generate code that interacts with the GCP core libraries at runtime. Fig. 2 shows the `json` for an application-specific speed type. The developer specifies the speed to use an unsigned byte with a range of 0 to 200km/h. In line 10, the correlation relationship states that, for this application, speed should not be found without both longitude and latitude. Line 11 shows a contextual relationship; the left side of `->` indicates the input parameters, and the right side describes the expression to be evaluated. This contextual relationship filters out any context that contains a speed over 150km/h when the context also indicates the user is in the Austin area. The complete GCP expression grammer is omitted for brevity.

Given a `json` input file, the GCP compiler creates a type class that inherits from and interacts with the core GCP libraries. Fig. 3 shows an example of generated code (in Python). The two `check` methods are created directly from the correlated and contextual constraints. These methods are used in the false positive detection process, which is invoked from within the core libraries.

GCP's API has three simple methods: ***serialize***, ***deserialize***, and ***query***. The ***serialize*** function takes the application's

```
{
  "type": {
    "name": "speed",
    "min": 0,
    "max": 200,
    "defaultValue": 0,
    "bits": 8,
    "signed": false
  },
  "correlated": ["longitude", "latitude"],
  "contextual": "value -> value <= 150 AND in(austin)",
  "var": {
    "austin:area": [30.2500, 97.7500, "10km"]
  }
}
```

Fig. 2: json input file

```
class SpeedType(SingleBitsSingleByteType):
    def __init__(self, value):
        SingleBitsSingleByteType.__init__(self)
        self.value = value
        self.name = 'speed'
        self.signed = false
        self.defaultValue = 0
        self.min = 0
        self.max = 200
        self.bits = 8
   def check_correlated(self, bf):
        t1 = bf('longitude'); t2 = bf('latitude')
        if (t1 and t2): return True
        return False
   def check_contextual(self, bf):
        l = current_location()
        if self.value <= 150 and in(l, austin): return True
        return False
```

Fig. 3: Generated Python code

context and creates a series of bytes that can be shared with directly connected peer devices. The ***deserialize*** function converts a series of (received) bytes into a queryable Bloomier filter, which is handed to the application.

The ***query*** function operates on an input Bloomier filter context summary and a predicate that can reference attributes potentially stored in the summary. A ***query*** invocation relies on the GCP core libraries: the query interpreter, the Bloomier filter context summary definitions, and the application-specified types. When an application invokes the ***query*** method, the interpreter parses the input predicate and determines whether the context summary contains the attributes referenced in the predicate. A ***query*** returns false if any of the necessary context attributes are not present. It returns true if the context attributes are present and the predicate evaluates to true. As part of the process of executing a ***query***, GCP invokes the application-specified false positive filters. These reside alongside the type definitions in the core libraries as a result of the automatic code generation process described previously. After applying these filters, only with very low probability does the ***query*** function return a false positive. Consider an example:

query(*contextSummary*, 'age of kid $\leq$ 7 *AND* TODAY'), where *contextSummary* is received from a neighboring mobile device. GCP interprets this predicate into the code in Fig 5.

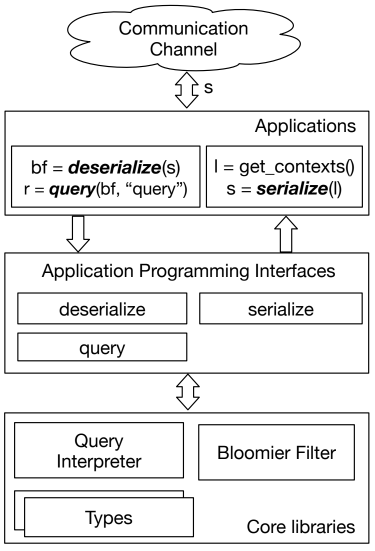

Fig. 4: Applications

```
x = contextSummary('age of kid');
y = contextSummary('date'); td = get_today()
if x and y: if x <= 7 and y == td: return True
return False
```

Fig. 5: Interpreted code

V. Conclusion

Representing context in a flexible and size efficient way is critical to mobile computing applications that need to share social, networking, and environmental situations using limited resources. The Grapevine Context Processor enables application developers to take advantage of properties specific to context to achieve large savings in the size of a context representation without sacrificing the quality of knowledge about that context. GCP uses novel data structures that provide a significant *performance* boost for mobile applications that require sharing context. Because those data structures result in false positives in context determination, GCP adds a false positive filter abstraction, which provides application programmers a simple and intuitive way to handle the complexities of the context summary structures. We codify this programming approach in a suite of automatic code generation tools and libraries, giving a very simple interface for developers to send and receive context summaries using the GCP framework.

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.

References

[1] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *CACM*, 1970.
[2] D. Charles and K. Chellapilla. Bloomier Filters: A Second Look. In *Proc. of ESA*, 2008.
[3] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal. The Bloomier filter: an efficient data structure for static support lookup tables. In *Proc. of SODA*, 2004.
[4] A. K. Dey and G. D. Abowd. Towards a Better Understanding of Context and Context-Awareness. In *Proc. of HUC*, 1999.
[5] M. Dietzfelbinger and R. Pagh. Succinct Data Structures for Retrieval and Approximate Membership. In *Proc. of ICALP*, 2008.
[6] C.-L. Fok, E. Grim, and C. Julien. Grapevine: Efficient situational awareness in pervasive computing environments. In *Proc. of PerCom WiP*, 2012.
[7] Gregory Hackmann, Christine Julien, Jamie Payton, and Gruia-Catalin Roman. Supporting Generalized Context Interactions. *SEM*, 3437(Chapter 8):91–106, 2004.
[8] C. Julien. The Context of Coordinating Groups in Dynamic Mobile Networks. In *Proc. of Coordination*, 2011.
[9] C. Julien, A. Petz, and E. Grim. Rethinking Context for Pervasive Computing: Adaptive Shared Perspectives. In *Proc. of ISPAN*, 2012.
[10] G J Nelson. *Context-aware and location systems*. PhD thesis, University of Cambridge, 1998.
[11] M. Perttunen, J. Riekki, and O. Lassila. Context Representation and Reasoning in Pervasive Computing: a Review. *IJMUE*, 2009.