D^{3}N: Programming Distributed Computation in Pocket Switched Networks

Eiko Yoneki, Ioannis Baltopoulos, and Jon Crowcroft
University of Cambridge, Computer Laboratory
Cambridge CB3 0FD, United Kingdom
{firstname.lastname}@cl.cam.ac.uk

ABSTRACT

We propose a novel approach to Pocket Switched Networks (PSNs) [8] using a specialised declarative language called 'D^{3}N'. A PSN is a recently devised type of communication based on physical proximity, where people encounter each other and their devices directly communicate within their communication range. D^{3}N allows us to program distributed applications based on reactive behaviour in a distributed set of nodes.

We exploit a functional language approach in designing D^{3}N for the clean abstraction given by pure declarative languages, at the same time, taking an advantage of well defined semantics. In this paper, we show a fragment of D^{3}N, describe the node runtime architecture, and illustrate its effectiveness through some examples.

Categories and Subject Descriptors

C.2.4 [**Computer Systems Organization**]: Computer Communication Networks—*Distributed Systems*; I.1.3 [**Computing Methodologies**]: Symbolic and Algebraic ManipulationLanguages and Systems

General Terms

Design, Languages, Algorithms

Keywords

Declarative Networking, Functional Programming, Distributed Computation, F♯, Delay Tolerant Networks

1. INTRODUCTION

The goal of this paper is to promote declarative networking as a simple programming paradigm for writing distributed applications over Pocket Switched Networks (PSN) [8]. In prior work we introduced the Haggle project [2], which explored a new communication paradigm: PSNs, a type of Delay Tolerant Networks (DTNs) [14]. DTNs provide communications in highly stressed environments with intermittent connectivity, variable delays and high error rates in decentralised and distributed environments over a multitude of devices that are dynamically networked. An important characteristic of DTNs is that they provide content storage as a core network service across applications. A partitioned network can deal with disconnected operations using store-and-forward type of operation. In PSNs people carry devices in their pockets, which communicate directly with other devices within their range or with infrastructure. As people move around, they carry messages with them and exchange them with nearby devices, resulting in an infrastructure-free, global, mesh network of devices. We present the design of a programming language that simplifies writing distributed applications over PSNs.

Our motivation looking at new programming environments is multi-faceted: devices in PSNs are typically mobile phones. Users need to be assured that new applications do not spend money on their behalf, on call-time or SMS for example; they need to be confident that private data is not compromised; and they need to understand the impact of applications on battery life and other resources. Declarative programming tools for model checking are reaching a level of maturity, and the community of programmers familiar with the functional approach is now large. PSNs represent a clean slate environment, where new approaches for building systems can be taken more easily. Finally, the software architecture devised in PSNs (and DTNs) is somewhat more complex as we will see in the rest of the introduction, and therefore it behoves us to take advantage of those tools that recent Computer Science provides.

In PSNs the network is the database. Each PSN node maintains 'data objects' annotated with 'metadata' in its data store persistently. When people seek information on shops, they want the answer itself, rather than a connection to shop web-sites. Thus, people directly search the network for content instead of reaching to a search engine looking for managed web content. A node updates its data store upon encountering other nodes in the network and exchanges data based on metadata matches. The forwarding mechanisms are driven by predicates/constraints derived from the dynamic network state [22], the observed network characteristics [15], or the logical network structure, such as social networks among device carriers [13], [25].

The current Haggle design is built over a layerless networking architecture that incorporates event-driven and asynchronous operations, which reside in the device as a kernel component. Functional components implement the functional logic and interact only directly with the kernel (see [20] for the further detail of Haggle implementation).

The reference implementation is developed over various platforms including Windows Mobile, Windows XP/Vista, Linux, Mac OS X, iPhone OS and Android [1] written in device dependent C/C++. Thus, when programmers write applications, they need

MobiHeld'09, August 17, 2009, Barcelona, Spain.

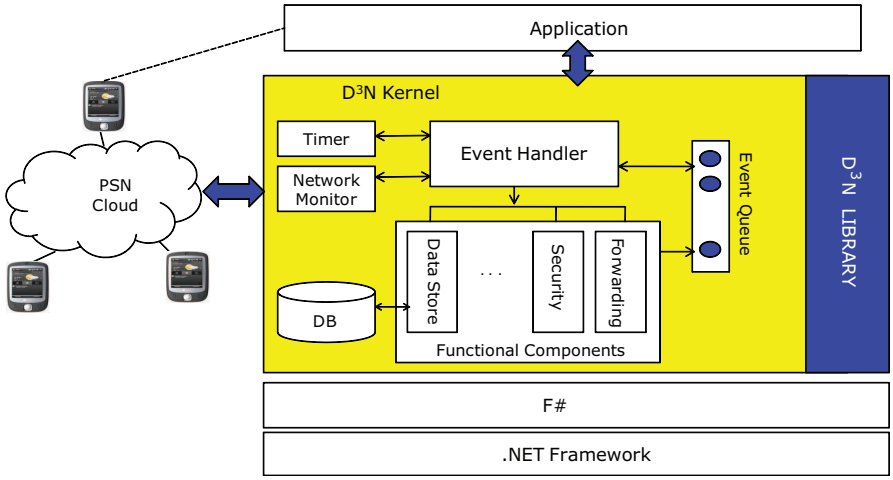

Figure 1: Overview of D^{3}N architecture

to implement them in C/C++ to interface to Haggle. Each device needs to have application executables for building distributed computation.

This complexity raises a question; what is a better way to program distributed computation in PSNs? We propose the declarative approach to program distributed computations in PSNs and call this system *Data-Driven Declarative Networking* (D^{3}N).

Our aim is to build a declarative network with an expressive language specific to PSNs. We exploit functional programming, the middle ground between pure declarative languages and imperative languages to define and implement D^{3}N. Declarative languages are too high level; they abstract too many implementation details like data flow and performance characteristics. On the other hand imperative languages are too low level, it is difficult to reason about programs and they are device dependent. Our approach simplifies the operation, robustness of the routing protocol, replication, overlay construction, indirection and so forth.

Starting from a standard functional programming language with concurrency primitives, we add data query and processing capabilities to the language while focusing on the distributed computations at each node. Algorithms for handling mobility and social connectivity observed at a node can be efficiently described in the declarative language. An important issue is to provide enough primitives to enable first-class processing of events within the language. The current reference implementation is written in F♯ [10] and not built on top of Haggle.

The rest of this paper is organised as follows: we describe the background of declarative networking and give a brief description of the node architecture Section 2; in Section 3, we give an overview of the language and demonstrate its use through examples in Section 4; we conclude and give directions for future work in Section 5.

2. DECLARATIVE NETWORKING IN D^{3}N

Declarative networking is a new idea in networking. Declarative languages make no distinction between code and data; they are used to define the desired functionality but not how such functionality is realised. For example, a search function can be described as 'what to look for' rather than 'how to look for' something. Additionally, the declarative approach abstracts the complexities that arise during networking and data processing. Concurrency and distribution pose algorithmic and implementation challenges; in a declarative setting, 'map/reduce' functions can be used to define such operations and to help build reliable distributed systems.

The P2 project [17, 21] introduces a revolutionary approach demonstrating how a declarative logic languages can be used for an overlay construction. The implementation, *Overlog*, is a derivative of *Datalog* and *Prolog* which can be used to describe an overlay network. The fact that 47 lines can describe CHORD [23] is impressive. The *Overlog* specification, does not include any operational process but takes a purely data driven approach. One of the criticisms on P2 is that there is no proof of semantic correctness of the described process.

In [9], Chu *et al.* introduce a Declarative Sensor Network (DSN), where the high-level declarative language is applied to data acquisition, dissemination and resource management, while retaining architectural flexibility. They demonstrate that a wide variety of ad-hoc sensor network protocols can be specified declaratively in a compact way.

Opis [11] is recent work, which takes a functional-reactive approach for developing distributed systems in OCaml. Thus, the aim of Opis is similar to D^{3}N. The LINQ project [16] extends the .NET Framework with language integrated operations for querying, storing and transforming data. Dryad [26] extends this to the wide area in a similar way to Google's Map-Reduce. However, given its large footprint and its target for connected operations, we feel a different approach is called for in PSNs. Our approach is using a functional style approach, which provides simple and clean semantics.

The D^{3}N system inherits its architecture from the current event-driven and modular Haggle architecture [20], [24]. Figure 1 highlights the core components of our system which consists of the D^{3}N kernel, the D^{3}N libraries and the D^{3}N language for writing applications. The kernel itself is built around the *Kernel Event Handler*, which communicates with *functional components* and *applications* through an *event queue*.

The data object is widely used in D^{3}N, which is the single format of information with the metadata. Data objects spread among D^{3}N nodes as they encounter each other in the network. They enter and leave a node through a single point. D^{3}N implements no layer-based architecture and communication between the functional components hence does not need to go through ordered layers. This leads to no distinction made between data objects received locally (from applications) or those received from other D^{3}N nodes in the network. Whether delivery is local or over the network is transpar-

ent.

The data store provides an interface that implements the primitives operations such as *search* described in Section 4.4. Every data object is timestamped and may age, which can provide efficient storage management for the devices with limited storage. Current implementation of Haggle uses SQLite [6] for the data store, which is sufficiently lightweight to run on resource constrained devices. It runs in a separate thread to prevent high latency I/O operations clogging the event queue. D^{3}N provides a subset of SQL for source level data queries.

Thus, applying a specialised declarative language (i.e. D^{3}N) over the described data centric plane is natural way to build a programming paradigm.

For the current implementation of D^{3}N we used F♯ [10], a concurrent and distributed functional programming language (variant of Standard ML [18]) over the .NET framework [19]. This enables application programmers to compile against D^{3}N libraries and make use of the entire Microsoft tool chain for developing, building, debugging and deploying their applications.

The D^{3}N language uses a combination of functional and declarative features offering several advantages: side-effects and state modifications are made explicit enabling easier reasoning about the code; functions are first class values and can be both the input and the result of computations; queries are first-class citizens in the language and are also strongly typed providing runtime safety. Extending this idea even further, queries over data can be expressed as higher-order functions that are applied in a distributed setting.

The D^{3}N library of primitive functions is built directly over F♯. Only a subset of F♯ is accessible to applications enforcing component and application writers to write modular code. We can further take advantage of recent work on F7 [7] to provide better checking of programmes before deployment, to assure people of reliability, security and safety properties of programs they may download to their smart phone without the exhaustive and expensive testing that has to be done with today's inadequate (C/C#) tools.

The Kernel Event Handler processes events from a shared data structure (event queue), where events are fed by applications and functional components. Functional components are part of D^{3}N, and they implement data forwarding, data storing and searching using primitive functions from the D^{3}N library as well as directly using F♯ and the .NET libraries. Communication between functional components is performed via events and asynchronous operations through the kernel. Component writers can write specialised components depending on the task at hand and register them as part of D^{3}N.

Example components are monitoring encountering nodes and various gossip-based data propagation mechanisms. The *Timer* and *Network Monitor* are special system components that run in an independent thread. Persistent storage for data objects is supported through a data store; each data object contains metadata, consisting of name and attribute pairs. Metadata operations like search, forwarding depend on the size and complexity of the network. Our design simplifies local data querying and processing mechanisms.

3. THE D^{3}N LANGUAGE

In this section we present the language available to application programmers, and give its informal semantics.

3.1 Syntax and Semantics

The starting point for the language design is a call-by-value, λ-calculus with concurrency extensions. We equip this language with query operations, first-class events and node identifiers. This is similar to the approach followed by the LINQ project [16], where .NET framework languages are provided with integrated operations for querying, storing and transforming data. Figure 2 contains the abstract syntax of the core calculus corresponding to D^{3}N, where bold words are keywords. A semantics for a language requires an evaluation rule for every language term. To simplify the presentation we syntactically divide the terms in fully evaluated values and unevaluated expressions. We assume a fixed collection of node identifiers. The terminals in the language consist of variables, a unit value (similar to void in object-oriented languages), primitive recursive functions and a constructor applied to a series of values.

a,b	::=	*Node identifiers*
M,N	::=	*Value*
	| x	variable
	| ()	unit
	| $\mu f.\lambda\tilde{x}.T$	recursive abstraction
	| $c\ \tilde{M}$	constructor application
S,T	::=	*Expression*
	| M	value
	| $M\ \tilde{N}$	application
	| $p\ \tilde{N}$	primitive application
	| **let** $x\ =\ S$ **in** T	let binding
	| **match** M **with** $c\ \tilde{x}$ **in** S **else** T	pattern matching
	| **fork** T	fork thread T
	| **send** $a\ M$	send value M to a
	| **receive** a	receive a message
	| **register** $M\ N$	register event handler
	| **select** M **from** ... **where** N	query

Figure 2: A core D^{3}N calculus syntax

We use the shorthand notation $\tilde{x}$ to mean $x_1, \ldots, x_n$, in abstractions and similarly in constructor applications. A program in D^{3}N is a closed expression S. To evaluate a program we proceed as follows:

- A term M is already fully evaluated.
- For a function application $M\ \tilde{N}$, ensure that M is an abstraction of the form $\mu f.\lambda\tilde{x}.T$; then substitute the value N_i for the variable x_i in T.
- The evaluation of a primitive function is provided by the kernel implementation; $p = \{\textbf{poll}, \ldots\}$.
- For a let binding, evaluate S until it yields a value M and then substitute M for the variable x in T.
- In a pattern match, unify the value M with the pattern $c\ \tilde{x}$. If M is a matching constructor with the same number of arguments $\tilde{N}$, then the whole match expression evaluates to S with the $\tilde{N}$ substituting the free variables $\tilde{x}$. If the unification fails the whole expression evaluates to T.
- To evaluate a fork expression, we evaluate T in a separate thread and return unit.
- A send expression, asynchronously sends the value M to the node a and returns unit.
- A receive expression expression blocks, listening for an event from the node a; once an event arrives, the call returns the message received.

- The primitive register, defines a function N to be called upon receipt of a message M.

- The select expression filters the elements

Many language features abundant in modern programming languages can be recovered as derived forms over the syntax of this calculus, and, hence, do not need to be taken as primitive. For example, integer numbers can be represented using the constructors **Zero** and **Succ** so that $2 = \textbf{Succ}(\textbf{Succ}(\textbf{Zero}))$. Similarly, characters can be mapped to integers using their ASCII value, and strings become lists of characters. Lists can be represented using the constructors **Nil** and **Cons** so that the list $[4,2,6]$ is represented as $\textbf{Cons}(4, \textbf{Cons}(2, \textbf{Cons}(6,\textbf{Nil})))$. The **if** ... **then** ... **else** ... construct, can be defined using **match**. Additional necessary functions like **map** (both sequential and parallel), **iter**, **filter**, **foldl** and **foldr** (similar to the reduce function in map-reduce) can be defined in the standard way. For the rest of the paper we assume these definitions exist.

3.2 Runtime System

The language relies on a small runtime system that is installed on all the devices that form a PSN. Each node is responsible for storing, indexing, searching, and delivering data. The current core runtime system (i.e. D^{3}N *Kernel*) consists of several functional components: *Kernel event Handler* (see Section 3.2.3), *Timer*, *Network Monitor* (see Section 3.2.1), *Data Store*, basic *Security*, and basic *Forwarding* (see Section 3.2.2) depicted in Figure 1. All primitive functions associated with the core D^{3}N calculus syntax described in Figure 2 are part of the runtime system. Each device-specific runtime system can be built with additional D^{3}N library functionality over the core D^{3}N runtime system. Among a collection of devices equipped with D^{3}N runtime systems, D^{3}N offers a programming paradigm for distributed computation.

PSN applications keep data and typically have a node-specific preference for data, described with metadata (i.e. data object). Data has a Time-To-Live (TTL) and is discarded when it expires. When a node receives some data, it checks if it matches the node's metadata and if so, it stores it and propagates the data to the other nodes. A search query has a TTL and until it expires it travels within the network. When the node has the matching data, it forwards the data. Each node gossips its own metadata when it meets other nodes. All these operations are implemented in the runtime system which is itself written in F♯.

3.2.1 Neighbourhood List

Every device maintains a list of dynamically discovered neighbours. The neighbourhood list contains the proximity of the device list with a timestamp of the last encounter. Periodically, the device updates the list of devices in its vicinity. When a new device is encountered, there is an exchange of neighbourhood lists. The primitive operation that enables the discovery process in the language is called **poll**$() : () \rightarrow$ **node list**. The **poll** function can be implemented various device discovery mechanisms (e.g. Bluetooth or WiFi).

3.2.2 Message Forwarding

Using the **poll**() function, **send** and **receive** primitives, we implement a unicast and multicast forwarding protocol. There are different algorithms for message forwarding with the most crude being epidemic; below we describe the default algorithm (Figure 3), which can be replaced by a user-defined one.

Each node has a unique node identifier and maintains a neighbourhood list of nodes that are currently in range, along with a routing table that is calculated from the neighbourhood lists of nearby nodes. Upon receiving a message with some new data, we check whether the data is destined to ourselves. If so we simply store the data locally. The message can contain operational code, which is run upon receipt, enabling mobile-agent like code migration. If the current node was not the destination for the data, we simply forward the data onwards to a selection of nodes (determined by the underlying PSN forwarding algorithm). If the current neighbourhood list is empty, we simply store the data along with a time-to-live (TTL) value, for future processing once we encounter new nodes.

```
match receive with
Data(d,n, unicast, TTL) as d →
  if n = nid then store d
  else if unicast then
    let nodes = lookup n routetbl in
    match nodes with
    | [] → store d
    | _ → iter (fun nd → send nd d) nodes
  else iter (fun nd → send nd d) poll()
```

Figure 3: Message forwarding

3.2.3 Kernel Event Handler

Event dispatching in the kernel event handler is implemented using an event queue where new events are appended. The event handler also implements TTL and event correlation mechanisms over the queue. The kernel maintains an association list between the event types and the functions that need to be called as a result of the event being dispatched. Figure 4 shows the event handling loop. When the eventHandler is called, it first de-references the event queue and attempts to pattern match it with the empty list. If the queue is indeed empty the function returns unit. In the case where the event queue was not empty, we split it in a head element (e) and a tail list (es). The head element is subsequently matched against the possible event types. Depending on the type of the event, we iterate over the corresponding function association list (fenc or fdep) calling each function in turn. The functions inside fenc and fdep, have been pre-registered using the **register** function. The function calls happen on the same thread.

```
type event = OnEncounter | OnDeparture | ...
let rec eventHandler () =
  match !queue with
  | [] → ()
  | e::es →
    match e with
    | OnEncounter → List.iter (fun f → f()) fenc
    | OnDeparture → List.iter (fun f → f()) fdep
    | ...
  eventHandler()
```

Figure 4: Kernel Event Handler

4. REALISTIC EXAMPLES USING D^{3}N

In this section we demonstrate a voting application, where voting takes a place among members in a social group. The group consists of members A, B, C, D, E, F and G. The group members could

```
type ballot = { locationA : int; locationB : int }
let emptyBallot = { locationA = 0; locationB = 0 };
let graph = getSocialGraph();
let voteForA():ballot = { locationA = 1; locationB = 0 }
let voteForB():ballot = { locationA = 0; locationB = 1 }

let rec smap f lst = // Sequential map
  match lst with
  | [] → []
  | n::ns → send f n;receive n :: smap f ns
let rec pmap f lst = // Parallel map
  match lst with
  | [] → []
  | n :: ns →
    fork (fun () →
      send f n;receive n
    ) :: pmap f ns
let rec reduce f se lst = // Reduce with starting element
  match lst with
  | [] → se
  | x::xs → f x (reduce f se xs)

let countVote (b1:ballot) (b2:ballot):ballot =
  { locationA = b1.locationA + b2.locationA;
    locationB = b1.locationB + b2.locationB }
let voteOfNode (node:string):ballot =
  match node with
  | "A" | "B" | "C" | "D" → voteForA()
  | "E" | "F" | "G" → voteForB()

reduce countVote emptyBallot (pmap voteOfNode graph)
```

Figure 5: Vote Application

be physically dispersed (e.g. in a large auditorium or in the campus) but find themselves in proximity range for PSN communication from time-to-time; no infrastructure based communication is available or it is too expensive. Now, they want to vote 'where to meet for dinner'. Every member in the group votes exactly once and the initiator of the program is the node A. All the members of the group are equipped with D^3N Kernel with D^3N Library and others around them have also D^3N capable devices forming a dynamically changing PSN topology.

4.1 Vote Application

The application functionality is built by mapping a vote function to the list containing B,C,D,E,F, and G and subsequently reducing the results to compute the final tally (Figure 5). An object containing the executable application code, initiator node id, and TTL is passed from A to the target nodes. Two types of **map** functions, i.e. sequential map (smap) and parallel map (pmap), are supported. The parallel map is used in the vote application and pmap operation breaks down 6 different communication tasks at the node A and processes them in parallel. Each node, upon receiving the object, executes the vote operation and sends back the result to the initiator. The root node A executes the **reduce** function when it receives all the results.

In the above example, when the data is sent from the node A to B, the actual forwarding algorithm depends on the available ones in D^3N. If the forwarding is pure epidemic, the data reaches B using an epidemic algorithm, where A sends the object to any node that it encounters. Node A could pass multiple nodes before reaching B. Alternative forwarding algorithms might require holding on to the object until the node A encounters the target node.

4.2 Cascaded Map

Knowledge of the logical network topology of members in a social group can be exploited in different deployments of the **map** function to program applications in an efficient way. In our previous work [13], we show a significant reduction in routing overheads by applying the knowledge of social network structure. The logical network topology could be obtained from various sources such as online social networks [3], [4], [5]. As an example, we describe how the **map** function can operate on the subset tree of nodes extracted from the social graph (i.e. cascade tree) and name this a *cascaded map*. The cascade tree is considered as a task graph, which indicates the order of operation. In the cascaded map operation, the **reduce** function is applied at the each node as explained in Section 4.3.

For example, the social graph is used to construct a minimum spanning tree with node A, the initiator, as the root node of the tree. At each node, the list of nodes for the map operation is extracted from the 'cascade tree'. Alternatively, the cascade tree can be recomputed from the social graph at each node. Therefore each node decides the target nodes for the map operation based on its 'cascade tree'. Using this approach the input to the map operation may express more advanced queries like: 'Given a node x, map on to connected nodes within two hops distance from x'. The children recursively compute a reduction step and continue the computation by sending the **map** function to their own children until a leaf node is reached.

4.3 Reduce for Cascaded Map

The cascaded map used for the voting program requires a more intelligent reduce operation at each node. Thus, the **reduce** function is sent to each node in the cascade tree prior to the cascaded map operation. Reduce operates at each node by aggregating the data that is received from other nodes along with the data that is saved in the local database. Depending on the position of the node in the cascade tree, a node behaves accordingly. If the node is a leaf in the tree, that means that it is the last node in the task graph, so it sends the aggregated results back to the initiator of the application. Otherwise, when the node is not a leaf node, it simply propagates the result towards the initiator node.

Because of the dynamic nature of PSNs, a node might encounter a node twice, or for that matter receive the data from another node multiple times. To ensure the correctness of the computation during a distributed reduction, a copy of each intermediate result is stored in the node.

Cascaded map is an example to build a D^3N library function and furthermore many other D^3N library can be built up. We will report the evaluation of various applications including the vote application in our future publication.

4.4 Search/Query

An important feature of the language is source level data queries. Queries are syntactically a restricted form of SQL queries for distributed data, stored on mobile devices. The following program searches the whole network for contact details of people working in the Computer Laboratory.

We express this query using a familiar notation, but note that queries are part of the source level syntax; they are not strings. Therefore they can be type-checked for correctness and are compiled to the primitive calculus that the node kernel recognises.

A selection query consists of a filtering function **where** over the nodes specified in the **from** part, and a mapping function that se-

lects the relevant fields from the returned records. This is similar to the general approach introduced by map-reduce [12]. Under this interpretation, select is merely a high-order function taking as inputs two functions (one for mapping, and one for filtering) and returning a collection of results.

In the examples the filtering function is compiled to **fun** $r \rightarrow$ **if** $r.company = \text{“}ComputerLaboratory\text{”}$ **then** Some(r) **else** None. It is a function that receives a record as input and checks if its company field is equal to the required string. We wrap the result in the Option type, to ensure that both branches of the **if** expression have the same type. Similarly, the mapping function in the first two examples gets compiled to **fun** r → r.name; it takes a record as input and returns the name field as a result.

The function in the third example is even simpler, it takes a record and returns it unmodified. After decomposing the query, we propagate the map and filter functions to other nodes and have the nodes evaluate them on their respective data stores and return the data back. Using the communication primitives that we presented previously, we send out the two functions along with the return node identifier for the responses.

```
select name from ∗
  where company=‘‘Computer Laboratory’’
select name from poll()
  where company=‘‘Computer Laboratory’’
select ∗ from ∗
  where company=‘‘Computer Laboratory’’
```

Figure 6: Source level queries

5. CONCLUSIONS AND FUTURE WORKS

In this paper, we introduce a declarative networking programming language for PSNs: D^{3}N, where communication resources are managed together with network connectivity. We provide an expressive language for building applications operating distributed computation. For implementing D^{3}N, we propose a functional language, instead of a declarative logic language, that provides an intermediate abstraction between implementation details and reasoning about logic flow. The current reference implementation is in F♯ targeting .NET platform taking advantage of a vast collection of .NET libraries for implementing D^{3}N primitives.

A direction for future research is to validate and verify the correctness of the design by implementing a compiler targeting various mobile devices. We aim to develop advanced analyses for programs written in our language like a type-and-effect system. An example could use pre-conditions and post-conditions on operations to prove that operations running on a local node respect the communication protocol with other distributed nodes.

Security issues are currently out of the scope of this paper. Executable code migrating from node to node, requires attention to potential security threats. We plan to approach this issue by integrating social network aspects, for example, restricting execution within a range of social topology.

Finally, we plan to put our code in public domain, so that potential application programmers can explore applications over D^{3}N.

Acknowledgements

We would like to acknowledge Steve Hand for critical reading and valuable comments. This research is funded in part by the EU Haggle project, IST-4-027918, and the SOCIALNETS project, 217141.

6. REFERENCES

[1] Google Android http://code.google.com/intl/fr-fr/android/.
[2] Haggle Project, http://www.haggleproject.org, 2008.
[3] Facebook http://www.facebook.com/, 2009.
[4] MySpace http://www.myspace.com/, 2009.
[5] Orkut http://www.orkut.com/, 2009.
[6] SQLite database engine. http://www.sqlite.org/, 2009.
[7] J. Bengtson, K. Bhargavan, C. Fournet, A. Gordon, and S. Maffeis. Refinement Types for Secure Implementations. In *Proc. CSF*, 2008.
[8] A. Chaintreau, P. Hui, J. Scott, R. Gass, J. Crowcroft, and C. Diot. Impact of human mobility on the design of opportunistic forwarding algorithms. In *Proc. INFOCOM*, 2006.
[9] D. Chu, L. Popa, A. Tavakoli, J. Hellerstein, P. Levis, S. Shenker, and I. Stoica. The design and implementation of a declarative sensor network system. In *Proc. SenSys*, 2007.
[10] D. Syme and J. Margetson. F♯: http://research.microsoft.com/fsharp. 2006.
[11] P. Dagand, D. Kostic, and V. Kuncak. Opis: Reliable distributed systems in ocaml. In *Proc. TLDI*, 2009.
[12] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proc. OSDI*, 2004.
[13] P. Hui, J. Crowcroft, and E. Yoneki. BUBBLE Rap: Social Based Forwarding in Delay Tolerant Networks. In *Proc. MobiHoc*, 2008.
[14] S. Jain, K. Fall, and R. Patra. Routing in a delay tolerant network. In *Proc. ACM SIGCOMM*, 2004.
[15] A. Lindgren, A. Doria, and O. Schelen. Probabilistic routing in intermittently connected networks. In *Proc. SAPIR*, 2004.
[16] LINQ Project. http://msdn.microsoft.com/en-gb/library/bb308959.aspx.
[17] B. Loo, P. Maniatis, and Others. Implementing Declarative Overlays. In *Proc. SOSP*, 2005.
[18] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Pressl, 1990.
[19] .NET Framework. http://msdn.microsoft.com/en-us/netframework/default.aspx.
[20] E. Nordström. *Challenged Networking: An Experimental Study of new Protocols and Architectures*. PhD thesis, 978-91-554-7239-9, Uppsala University, 2008.
[21] P2. http://p2.berkeley.intel-research.net/.
[22] T. Spyropoulos, K. Psounis, and C. Raghavendra. Spray and wait: An efficient routing scheme for intermittently connected mobile n etworks. In *Proc. WDTN*, 2005.
[23] I. Stoica, R. Morris, D. Liben-Nowell, D. Kargerz, M. Kaashoekz, F. Dabekz, and H. Balakrishnan. Chord: A peer to peer lookup protocol for internet applications. *IEEE/ACM Trans. on Networking*, 11, 2004.
[24] J. Su, J. Scott, P. Hui, J. Crowcroft, E. de Lara, C. Diot, A. Goel, M. Lim, and E. Upton. Haggle: Seamless networking for mobile applications. In *UbiComp*, 2007.
[25] E. Yoneki, P. Hui, S. Chan, and J. Crowcroft. A Socio-Aware Overlay for Multi-Point Asynchronous Communication in Delay Tolerant Networks. In *Proc. MSWiM*, 2007.
[26] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *Proc. OSDI*, 2008.