# Implementing Network Protocols as Distributed Logic Programs

Boon Thau Loo

University of Pennsylvania

boonloo@cis.upenn.edu

*Declarative networking* [2, 4, 3, 1] is an application of database query-language and processing techniques to the domain of networking. Declarative networking is based on the observation that network protocols deal at their core with computing and maintaining distributed state (e.g., routes, sessions, performance statistics) according to basic information locally available at each node (e.g., neighbor tables, link measurements, local clocks) while enforcing constraints such as local routing policies. Recursive query languages studied in the deductive database literature [6] are a natural fit for expressing the relationship between base data, derived data, and the associated constraints. Simple extensions to these languages and their implementations enable the natural expression and efficient execution of network protocols.

Declarative networking aims to accelerate the process of specifying, implementing, experimenting with and evolving designs for network architectures. Declarative networking can reduce program sizes of distributed protocols by orders of magnitude relative to traditional approaches. In addition to serving as a platform for rapid prototyping of network protocols, declarative networking also open up opportunities for automatic protocol optimization and hybridization, program checking and debugging.

This paper presents an introduction to declarative networking using a simple routing protocol example. For more details on declarative networking related projects, refer to the NetDB@Penn website [5], and the RapidNet [7] declarative networking engine.

## Declarative Networking By Example

The Network Datalog (*NDlog*) language used in declarative networking is based on extensions to traditional Datalog, a well-known recursive query language designed for querying graph-structured data in a centralized database. *NDlog*'s integration of networking and logic is unique from the perspectives of both domains. As a network protocol language, it is notable for the absence of any communication primitives like "send" or "receive"; instead, communication is implicit in a simple high-level specification of data partitioning. In comparison to traditional logic languages, it is enhanced to capture typical network realities including distribution, link-layer constraints on communication (and hence deduction), and soft-state semantics.

We introduce Network Datalog (*NDlog*), the declarative network using an example program shown below that implements the *Path-vector protocol*, which computes in a distributed fashion, for every node, the shortest paths to all other nodes in a network. The path-vector protocol is used as the base routing protocol for exchanging routes among Internet Service Providers.

```
sp1 path(@Src,Dest,Path,Cost) :- link(@Src,Dest,Cost),
                                 Path = f_init(Src,Dest).
sp2 path(@Src,Dest,Path,Cost) :- link(@Src,Nxt,Cost1),
                                 path(@Nxt,Dest,Path2,Cost2),
                                 Cost = Cost1+Cost2,
                                 Path = f_concatPath(Src,Path2).
sp3 spCost(@Src,Dest,min<Cost>) :- path(@Src,Dest,Path,Cost).
sp4 shortestPath(@Src,Dest,Path,Cost) :- spCost(@Src,Dest,Cost),
                                         path(@Src,Dest,Path,Cost).
Query shortestPath(@Src,Dest,Path,Cost).
```

The program has four rules (which for convenience we label `sp1-sp4`), and takes as input a base (*extensional*) relation `link(Src, Dest, Cost)`. Rules `sp1-sp2` are used to derive "paths" in the graph, represented as tuples in the derived (*intensional*) relation `path(Src,Dest,Path,Cost)`. The `Src` and `Dest` fields represent the source and destination endpoints of the path, `Path` is the actual path from `Src` to node `Dest`. The number and types of fields in relations are inferred from their (consistent) use in the program's rules.

Since network protocols are typically computations over distributed network state, one of the important requirements of *NDlog* is the ability to support rules that express distributed computations. *NDlog* builds upon traditional Datalog by providing control over the storage location of tuples explicitly in the syntax via *location specifiers*. Each location specifier is a field within a predicate that dictates the partitioning of the table. To illustrate, in the above program, each predicate has an "`@`" symbol prepended to a single field denoting the location specifier. Each tuple generated is stored at the address determined by its location specifier. For example, each `path` and `link` tuple is stored at the address held in its first field `@Src`.

Rule `sp1` produces `path` tuples directly from existing `link` tuples, and rule `sp2` recursively produces `path` tuples of increasing cost by matching (joining) the destination fields of existing links to the source fields of previously computed paths. The matching is expressed using the repeated `Nxt` variable in `link(Src,Nxt,Cost1)` and `path(Nxt,Dest,Path2,Cost2)` of rule `sp2`. Intuitively, rule `sp2` says that "if there is a link from node `Src` to node `Nxt`, and there is a path from node `Nxt` to node `Dest` along a path `Path2`, then there is a path `Path` from node `Src` to node `Dest` where `Path` is computed by prepending `Src` to `Path2`". The matching of the common `Nxt` variable in `link` and `path` corresponds to a *join* operation used in relational databases.

Given the `path` relation, rule `sp3` derives the relation `spCost(Src,Dest,Cost)` that computes the minimum cost `Cost` for each source and destination for all input paths. Rule `sp4` takes as input `spCost` and `path` tuples and then finds

`shortestPath(Src,Dest,Path,Cost)` tuples that contain the shortest path `Path` from `Src` to `Dest` with cost `Cost`. Last, as denoted by the `Query` label, the `shortestPath` table is the output of interest.

## Shortest Path Execution Example

We step through an execution of the *shortest-path NDlog* program above to illustrate derivation and communication of tuples as the program is computed. We make use of the example network in Figure 1. Our discussion is necessarily informal since we have not yet presented our distributed implementation strategies; in the next section, we show in greater detail the steps required to generate the execution plan. Here, we focus on a high-level understanding of the data movement in the network during query processing.

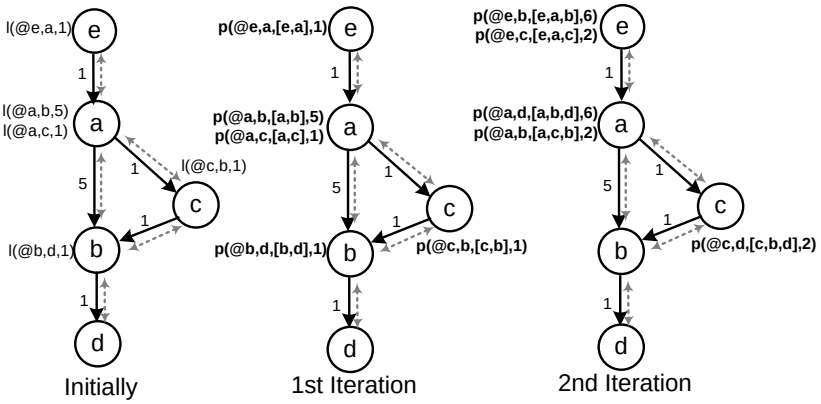


**Fig. 1.** Nodes in the network are running the shortest-path program. We only show newly derived tuples at each iteration.

For ease of exposition, we will describe communication in synchronized *iterations*, where at each iteration, each network node generates *paths* of increasing hop count, and then propagates these paths to neighbor nodes along links. We show only the derived paths communicated along the solid lines. In actual query execution, derived tuples can be sent along the bidirectional network links (dashed links).

In the $1^{st}$ iteration, all nodes initialize their local path tables to 1-hop paths using rule `sp1`. In the $2^{nd}$ iteration, using rule `sp2`, each node takes the input paths generated in the previous iteration, and computes 2-hop paths, which are then propagated to its neighbors. For example, `path(@a,d,[a,b,d],6)` is generated at node *b* using `path(@b,d,[b,d],1)` from the $1^{st}$ iteration, and propagated to node *a*. In fact, many network protocols propagate only the *nextHop* and avoid sending the entire path vector.

As paths are computed, the shortest one is incrementally updated. For example, node *a* computes the cost of the shortest path from `a` to `b` as 5 with rule `sp3`,

and then finds the corresponding shortest path `[a,b]` with rule `sp4`. In the next iteration, node *a* receives `path(@a,b,[a,c,b],2)` from node `c`, which has lower cost compared to the previous lowest cost of 5, and hence `shortestPath(@a,b,[a,c,b],2)` replaces the previous tuple (the first two fields of source and destination are the primary key of this relation).

Interestingly, while *NDlog* is a language to describe networks, there are no explicit communication primitives. All communication is implicitly generated during rule execution as a result of data placement specifications. For example, in rule `sp2`, the `path` and `link` predicates have different location specifiers, and in order to execute the rule body of `sp2` based on their matching fields, `link` and `path` tuples have to be shipped in the network. It is the movement of these tuples that generates the messages for the resulting network protocol.

## References


1. B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative Networking: Language, Execution and Optimization. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, 2006.
2. B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative Networking. In *Communications of the ACM (CACM)*, 2009.
3. B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing Declarative Overlays. In *Proceedings of ACM Symposium on Operating Systems Principles*, 2005.
4. B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative Routing: Extensible Routing with Declarative Queries. In *Proceedings of ACM SIGCOMM Conference on Data Communication*, 2005.
5. NetDB@Penn. `http://netdb.cis.upenn.edu/`.
6. R. Ramakrishnan and J. D. Ullman. A Survey of Research on Deductive Database Systems. *Journal of Logic Programming*, 23(2):125–149, 1993.
7. RapidNet Declarative Networking Engine. `http://netdb.cis.upenn.edu/rapidnet/`.