Distributed Graph-store Processing

Eric Griffis*

May 30, 2013

Abstract

Social computing is the branch of computer science that studies the human aspects of emerging computing paradigms like social networking, but many popular Internet-based technologies with strong social components share significant overlap in their conceptual models and implementation details. In this report, I define a subset of distributed computing systems called *social computing systems* and then present a novel general-purpose social computing platform.

1 Introduction

Strictly speaking, social computing proper [15] is the branch of computer science that studies the properties and social implications of connective, collaborative, community-based computing paradigms like social networking. While existing social computing research is concerned with the behavior of human participants in social computing tasks such as collaborative filtering, online auctions, and reputation systems, many popular Internet-based technologies with strong social components share a remarkable amount of overlap in their conceptual models and implementation details.

In this report, I define *social computing systems* as the subset of distributed computing systems that emphasize the timely flow of information between agents with potentially competing interests. Existing social computing systems include the World Wide Web, the global e-mail delivery system, and even a distributed social networking platform[1]. In a social computing system, nodes exchange information as structured messages. For instance, an HTTP request is a sequence of key-value pairs, as are SMTP headers and message envelopes.

Social computing systems are particularly amenable to graph models as, e.g., communication networks induced by message passing. Even HTTP and SMTP protocol messages can be modeled as association lists—a simple form of graph—because of their key-value link structure. Dynamic graphs [14], or graphs that may change over time, provide a theoretical basis for modeling real-world complex dynamic networks [8] such as social and communication networks.

In light of the inherent graph structure of common social computing tasks, I decided to study how far the graph metaphor extends within the context of social computing systems. Graph analysis is clearly appropriate for querying, or otherwise probing the structural properties of, induced networks such as the aforementioned communications networks, but what are the trade-offs of adopting the graph model for more mundane aspects like messages? Complementarily, what about for more abstract aspects like raw computation? Can we quantify the trade-offs of heavily graph-based systems with respect to existing solutions? Might we broaden the class of software systems that

*UCLA CS 246, Winter 2013

[1] http://diasporaproject.org/

benefit from such an approach? In this report, I address these issues informally, within the context of the *Distributed Graph-store Processing* (DGrasp) platform, a novel social computing platform designed to leverage graph models whenever possible.

This report is organized as follows. Section 2 motivates the utility of social computing systems by comparing DGrasp to the existing body of related work. Section 3 gives an informal description of the DGrasp social computing platform, and sections 4 and 5 detail the case study and implementation used to derive the DGrasp platform design. Section 6 presents an informal analysis of the platform design. Section 7 concludes.

2 Related Work

The technical merits of coordinating distributed computations as dynamic graphs have been studied since at least the Eighties [10], and work on graph-based models for general computation [11] began in the Seventies. More recently, interest in the Semantic Web has promoted research into graph-structured message passing [16] as the basis for cutting edge knowledge representation and sharing systems. In contrast to more "traditional" distributed computing paradigms, social computing systems do not necessarily focus on the presentation of massive-scale raw computing resources as a single virtual entity as in grid, cluster, or participatory computing systems. How, then, might we focus social computing research efforts?

I believe that issues relevant to social computing are, as the name suggests, of a fundamentally social nature. There is certainly no shortage in recent news reports of privacy-motivated scandals perpetrated by governments and social network operators, but social computing systems give rise to a host of more subtle problems that reflect real-life social issues [17] [9]. To illustrate, consider the disparities between old and new social computing systems. Older systems like the Web and e-mail tend to be fully distributed—ignoring the subtleties of modern IP routing, anyone with Internet access may operate Web or e-mail servers. Although newer systems like Facebook and Twitter leverage computer clusters and content delivery networks internally, these networks tend to be owned and operated by a single central entity—the average social network user has little hope for operating their own network. There is an unfortunate irony in the fact that the Web—one of world's most successful fully distributed software systems—gave rise to so many fully centralized services.

To remain competitive, centralized social networking services require massive computing resources which are almost universally paid for by advertising revenue. Research on existing virtual worlds like Second Life and World of Warcraft partition competing agents as motivated by corporate, government, or citizen interests [17]. Clearly, a generic and fully distributed computational model that admits heterogeneous ownership and operation of participating resources would shift the burden of responsibility and control of information away from the former and toward the latter. Social computing systems are, in this sense, a form of digital democracy. Moreover, a fully distributed system can deliver strong privacy guarantees to participants not possible in a centralized approach [9].

A standard platform for heterogeneously owned and operated distributed systems that reduces dependence on complex code and physical resources would dramatically reduce the existing barriers to entry to social computing tasks for the average user. Two additional advantages to such systems is the interoperability and performance gained by reducing dependence on low-level parsing tasks. For instance, media file formats are designed to carry metadata for media-consuming applications to extract and process on demand. A system with native structured messages could extract and intern the metadata only once as part of an "import" procedure and share the extracted information

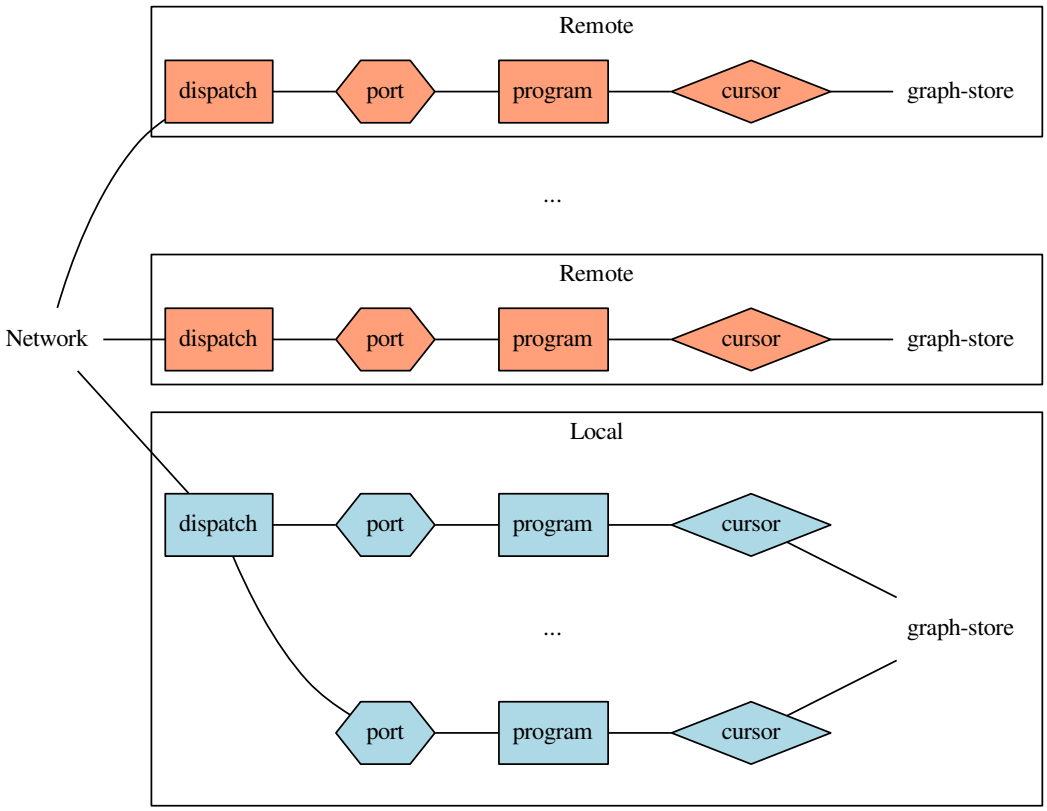

Figure 1: Local system architecture

directly via structured message passing.

Fortunately, graph transformation and graph rewriting systems are well-studied in the literature and can be shown to possess nice properties [7] [3] [12] [5] [18] [6] [2] [4]. Most notably, properly-designed graph rewrite systems can be made sound, complete, confluent, and so on.

3 System Design

In order to discuss the details of the DGrasp social computing platform, we must first establish some basic definitions and notation. Figure 1 outlines the local DGrasp system architecture. A *graph-store* is essentially a structured memory manager and persistent storage mechanism. It operates like a black box that translates between nodes and node references. A *node* is a set of rewrite rules that contains its properties and edges. Primitive graph-store operations do not distinguish between properties and edges because edges are treated as bindings from edge labels to *node references*, or opaque tokens that each identify a distinct node within the graph-store. A *program* is a graph, rooted at a particular node, that operates on a cursor and at least one port. Programs are drawn as boxes. A *process* is an active in-memory copy of some program. Note that other sections of this report frequently use the terms node, program, and message interchangeably.

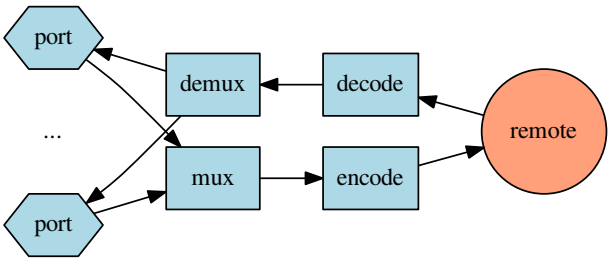

Figure 2: The dispatch architecture

A *cursor* is an opaque token that serves as the interface between a process and its local graph-store. Cursors are drawn as diamonds. When given a node, the cursor stores a copy of the node in the graph-store and returns a reference to the stored copy. When given a node reference, the cursor produces a copy of the referenced node in memory. When given both a node and a node reference, the cursor replaces the stored copy of the referenced node with a copy of the given node.

Each DGraph instance manages its own *dispatch* subsystem, from which processes, cursors, and ports are manufactured. A *port* is an opaque token that serves as the interface between processes not necessarily running within the same DGrasp instance. Ports are drawn as hexagons. Unlike conventional TCP and UDP sockets, which are identified by a port number, DGrasp ports may be identified by arbitrary tokens, e.g., "HTTP" or "IMAP." There are two kinds of ports. A *client port* represents the initiating side of a two-way communication channel, while a *server port* represents the passive side.

The DGrasp platform implements distributed computations as *tasks*, or dynamic compositions of processes that pipe information within and between distinct DGrasp instances. Programs may register with the local dispatch their intention to handle server or client ports. The dispatch then initiates all client port handler processes with a client port and all server port handler processes with a server port. Processes may register new server port handlers or request additional client ports at any time.

3.1 Important Details

The DGrasp platform design is based on the observation that complex social computing tasks may be decomposed into relatively simple dynamic information flows. For example, Web browser-server interactions is just structured message passing for which response messages may contain arbitrary binary blobs, and e-mail delivery involves finite-length chains of essentially the same process run in reverse.

Figure 2 shows the dispatch subsystem's high-level architecture. Dispatch may be simpler than the figure suggests because multiplexing and demultiplexing operate directly on messages as graphs, before (after) any transport encoding (decoding) occurs. This arrangement allows DGrasp to abstract away the network transport implementation details completely.

4 Case Study: a Simple Web Server and Client

Despite the Web being perhaps the most obvious candidate for a simple social computing system, discussion of DGrasp programs requires more detailed notation. Since space is limited, I describe informally the details of the simple Web server program diagram in figure 4 and defer rigorous specification to future work.

In figure 4, literal tokens appear as orange boxes. Port and cursor actions appear as green hexagons and diamonds, respectively, with the action name in bold. Semicolons denote sequences of operations. Nodes are denoted by simple shapes, large gray areas, or matched curly brace pairs that link to bindings denoted by equals signs that in turn link to a left- and a right-hand child. The structure on the left-hand side is a pattern to be matched within the evaluation context of the containing node. The blue nodes with italicized names are variables that name the sub-graph whose root coincides with the variable. Matching structures are replaced by their corresponding target structure on the right-hand side of the equals sign, with all free occurrences of pattern variables substituted as in traditional functional pattern matching. A period denotes the evaluation context of a node. The blue diamond that contains an underscore is a special pattern that matches, without naming, any structure. Note that this program is recursive: after each request, the web server is re-invoked under the assumption that the port will block until the next request is available.

We may interpret this diagram as follows:

1. The `web_sever` program consumes a cursor and a port, then blocks until a request message arrives on the port.

2. When the request contains `GET` followed by a path, the server loads the node in the local graph-store that corresponds to the path, sends the node through the port, and waits for the next request.

3. When the request contains `PUT` followed by a path and a target structure, the server stores the target in the local graph-store at the given path, sends an `OK` message through the port, and waits for the next request.

4. For request of any other form, the server sends a `BAD_REQUEST` message though the port and then waits for the next request.

Figures 5 and 6 together constitute the complete simple web client, where the latter extends the former with multiple pipe-lined requests. The interpretation of figure 5 is straight-forward: the `web_client` program consumes a cursor and a port and then returns a node that contains the rules for non-pipelined document retrieval tasks. The ellipsis indicates placement of the details covered by figure 6. This program differs from the web server in that it simply produces an evaluation context without any direct evaluation. This difference is due to the fact that `web_server` is a complete, self-contained program, while `web_client` is more like a library function for other programs that wish to retrieve Web documents. To fetch the document, a program simply evaluates a message of the form `HTTP GET some-path` within the evaluation context returned by `web_client`.

While programs may re-use a context generated by `web_client` to manually pipeline requests, such functionality is desirable in general, and so I capture a simple version of Web request pipelining in figure 6. This program sends all of the requests in the linked list `requests` before receiving any responses. For each request sent, the same `responder` message is used as an evaluation context to process the corresponding response.

For comparison, here are example textual renderings of the diagrams in figures 4, 5, and 6:

```
web_server :cursor :port =
{ HTTP (GET :path) =
  ( port send (OK (cursor load path))
  ; web_server cursor port
  )
; HTTP (PUT :path :target) =
  ( cursor store path target
  ; port send OK
  ; web_server cursor port
  )
; HTTP _ =
  ( port send BAD_REQUEST
  ; web_server cursor port
  )
}.( HTTP (port recv) )

web_client :cursor :port =
{ HTTP :request =
  ( port send request
  ; port recv
  )
; HTTP_PIPELINE :requests :responder =
  { write-loop (:request ; :rest) =
    ( port send request
    ; write-loop rest
    )
  ; write-loop :request =
    ( port send request
    ; read-loop requests
    )
  ; read-loop (:request ; :rest) =
    ( responder (port recv)
    ; read-loop rest
    )
  ; read-loop :request = responder (port recv)
  }.( write-loop requests )
}
```

5 Experiment

The project started with nothing more than a clear vision: to design a generic platform for social computing systems. Previous work into graph-based computational models fed my intuition that such models were a good fit for some interesting class of distributed computing systems that involved structured message passing, so I built a simple tool to simulate remote graph-store management operations via structured messages. The high-level implementation details are given in figure 3. Chief insights gained from this experiment were the fact that the chosen concepts indeed fit together well enough to do meaningful work, along with a sandbox in which to explore related projects.

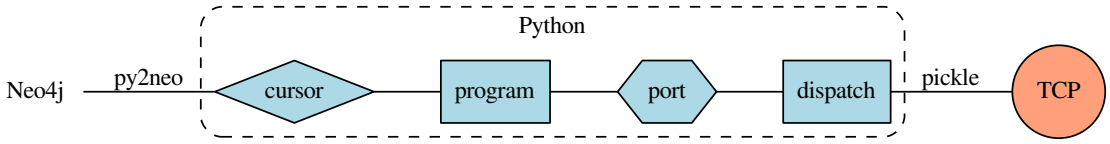

Figure 3: Implementation overview

The graph-store was implemented with Neo4j, a popular property graph database [1] written in Java. Very simple message passing programs were written in Python, which connected to Neo4j via the py2neo library and to remote instances via TCP sockets. Nodes were modeled as Python dictionaries, and the encode/decode process was hard-coded into each program via the standard Python object pickling facilities. Multiplexing was not implemented. This crude implementation was sufficient to derive the complete DGrasp design.

6 Evaluation

Not surprisingly, graphs are a remarkably flexible medium for capturing the structures and activities relevant to social computing systems. Furthermore, because DGrasp models everything from local computations to distributed tasks as graph transformations, the system exhibits an unprecedented level of *homoiconicity* which not only admits rich metaprogramming features like macros and REPL-style interactions, but might also suggest novel combinations of metaprogramming features.

The simple message-passing implementation gave me enough insight into the complete DGrasp platform design to discover that the distinction between push- and pull-based protocols is reduced to a matter of policy, as opposed to a defining characteristic of the task at hand. I had originally planned three case studies: Web, e-mail, and subscription-based chat; but the latter two were ultimately obviated by the former. Specifically, a simple e-mail delivery system can be implemented as a client-server program pair such that the roles of the two are swapped, along with a simple routing policy implementation that determines whether to deliver, forward, or reject each incoming request. A simple chat system can be implemented similarly by adding a "virtual circuit" matrix to the routing policy implementation to track subscriptions. In fact, arrangements such as these e-mail and chat networks constitute a crude form of onion routing [13], a flexible and efficient message routing protocol with strong privacy-preserving properties.

An interesting property of the DGrasp platform is that pipelining may occur at the message level—between the application and network levels. Consequently, DGrasp can realize amortized connection overhead savings not possible in distributed computing system combinations. Since efficient distributed social computing systems inherently prefer interaction with a small set of trusted remote systems, we may reasonably expect the savings to be substantial.

We see similar savings through the practice of "metadata-import" as outlined in section 2. Savings from metadata extraction and transformed into a format more conducive to distributed graph-store processing (i.e., structured message passing) may potentially be realized by each participant in, and at each step of, tasks such as media processing and sharing. Because the modern Web is essentially a giant media processing and sharing platform, we may reasonably expect significant overall savings.

While the DGrasp platform design exhibits remarkable qualities in terms of flexibility, its raw performance characteristics remain undiscovered. Before DGrasp can be deployed in a production setting, it must be subject to rigorous analyses to determine suitability for varying levels of detail. The experiment detailed in this report demonstrates that DGrasp is at least capable of handling extremely simple Web or messaging applications, but nothing more. Application of existing work on graphs, including dynamic graphs and graph transformations, should prove useful for illuminating any hard bounds on performance.

7 Conclusion

In this report, I identified the *social computing* class of distributed systems and motivated the informal design of DGrasp, a novel social computing platform, through a case study and a simple graph-based message passing experiment. Despite the simplicity apparent in the design of the DGrasp platform, architecting general purpose distributed computing platforms is no small feat. In particular, choosing simple abstractions that enhance flexibility without obviously sacrificing performance can be overwhelming due to the sheer number of possibilities. Indeed, a significant portion of this project was dedicated to comprehensive literature survey to determine that social computing was an appropriate context in which to explore distributed computing platforms of generality sufficient enough to capture a broad range of useful applications. Fortunately, critical insights were achieved early enough in the project to make substantial progress on the design. Most notably, I realized the duality of nodes, programs, and messages.

Though the unusual flexibility of distributed graph-store processing stands as compelling evidence of the utility of such systems, much work remains before the paradigm can be considered of practical use. Specifically, I would like to pursue rigorous analysis of the performance characteristics of DGrasp in particular and social computing platforms in general, as well as a proper formalism for social computing systems. Indeed, I expect to make progress along these lines over the upcoming academic year.

References

[1] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Computing Surveys (CSUR)*, 40(1):1, 2008.

[2] Zena M Ariola and Jan Willem Klop. Equational term graph rewriting. *Fundamenta Informaticae*, 26(3):207–240, 1996.

[3] Hendrik Pieter Barendregt, Marko CJD van Eekelen, John RW Glauert, J Richard Kennaway, Marinus J Plasmeijer, and M Ronan Sleep. Term graph rewriting. In *PARLE Parallel Architectures and Languages Europe*, page 141–158, 1987.

[4] Erik Barendsen and Sjaak Smetsers. Conventional and uniqueness typing in graph rewrite systems. In *Foundations of Software Technology and Theoretical Computer Science*, page 41–51, 1993.

[5] Stefanus Cornelis Christoffel Blom. Term graph rewriting. syntax and semantics. 2001.

[6] Dorothea Blostein, Hoda Fahmy, and Ann Grbavec. Issues in the practical use of graph rewriting. In *Graph Grammars and Their Application to Computer Science*, page 38–55, 1996.

[7] TH Brus, Marko CJD van Eekelen, MO Van Leer, and Marinus J Plasmeijer. Clean—a language for functional graph rewriting. In *Functional Programming Languages and Computer Architecture*, page 364–384, 1987.

[8] Arnaud Casteigts, Paola Flocchini, Walter Quattrociocchi, and Nicola Santoro. Time-varying graphs and dynamic networks. *International Journal of Parallel, Emergent and Distributed Systems*, 27(5):387–408, 2012.

[9] Leudo Antonio Cutillo, Refik Molva, and Thorsten Strufe. Privacy preserving social networking through decentralization. In *Wireless On-Demand Network Systems and Services, 2009. WONS 2009. Sixth International Conference on*, page 145–152, 2009.

[10] Pierpaolo Degano and Ugo Montanari. A model for distributed systems based on graph rewriting. *Journal of the ACM (JACM)*, 34(2):411–449, 1987.

[11] Hartmut Ehrig. Introduction to the algebraic theory of graph grammars (a survey). In *Graph-Grammars and Their Application to Computer Science and Biology*, page 1–69, 1979.

[12] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of algebraic graph transformation*, volume 373. Springer Heidelberg, 2006.

[13] David Goldschlag, Michael Reed, and Paul Syverson. Onion routing. *Communications of the ACM*, 42(2):39–41, 1999.

[14] F. Harary and G. Gupta. Dynamic graph models. *Mathematical and Computer Modelling*, 25(7):79–87, 1997.

[15] Irwin King, Jiexing Li, and Kam Tong Chan. A brief survey of computational approaches in social computing. In *Neural Networks, 2009. IJCNN 2009. International Joint Conference on*, page 1625–1632, 2009.

[16] Graham Klyne, Jeremy J. Carroll, and Brian McBride. Resource description framework (RDF): concepts and abstract syntax. *W3C recommendation*, 10, 2004.

[17] Brian E Mennecke, David McNeill, Matthew Ganis, Edward M Roche, David A Bray, Benn Konsynski, Anthony M Townsend, and John Lester. Second life and other virtual worlds: A roadmap for research. *Communications of the Association for Information Systems*, 22(20):371–388, 2008.

[18] Grzegorz Rozenberg and Hartmut Ehrig. *Handbook of graph grammars and computing by graph transformation*, volume 1. World Scientific London, 1999.

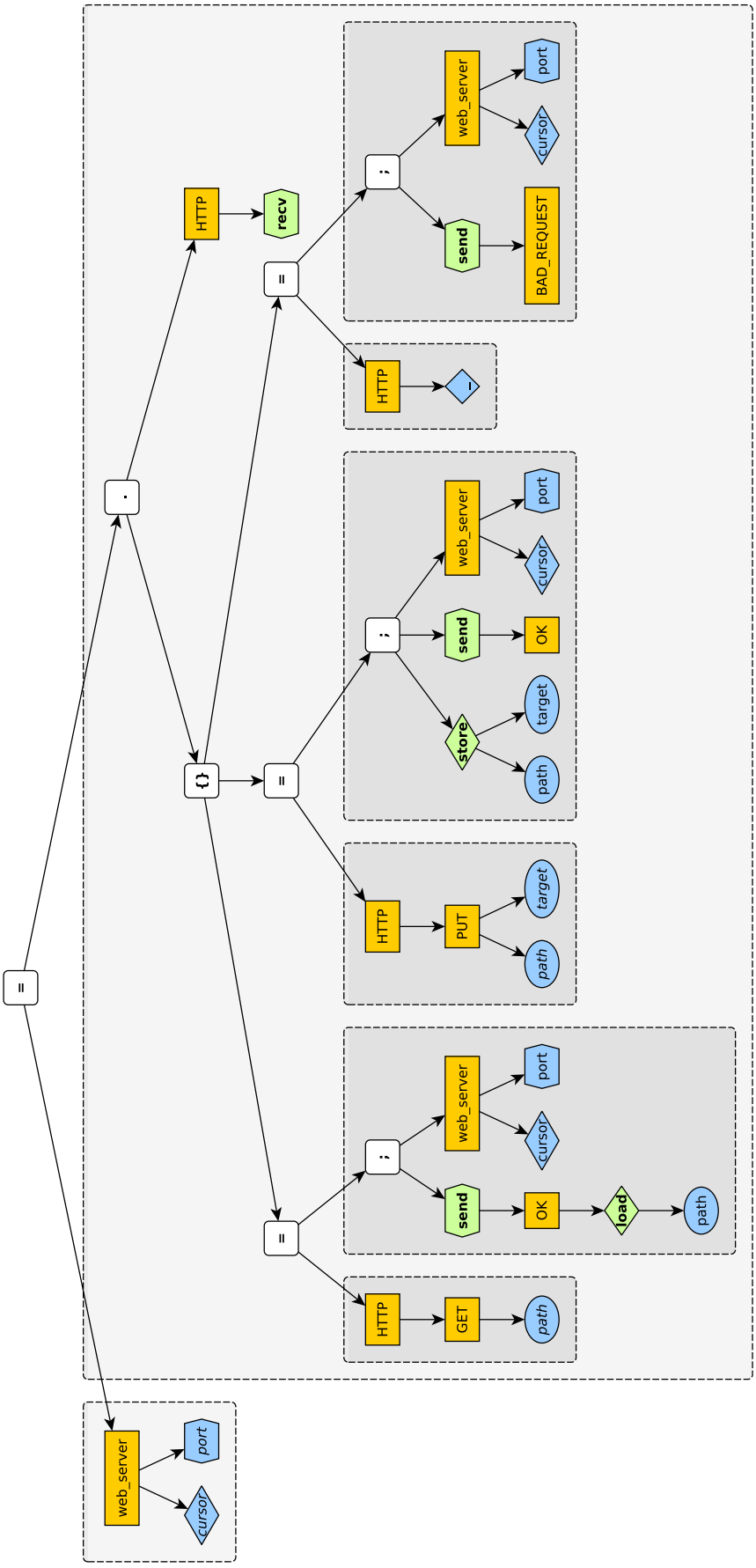

Figure 4: A simple web server

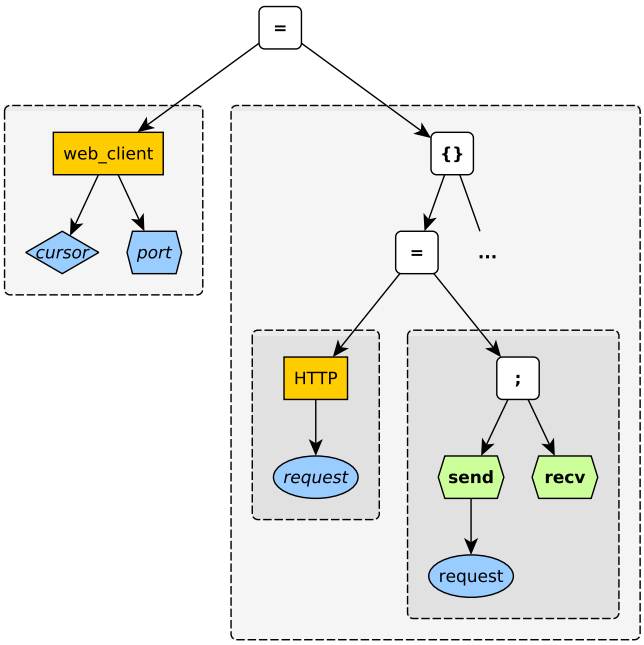

Figure 5: A simple web client

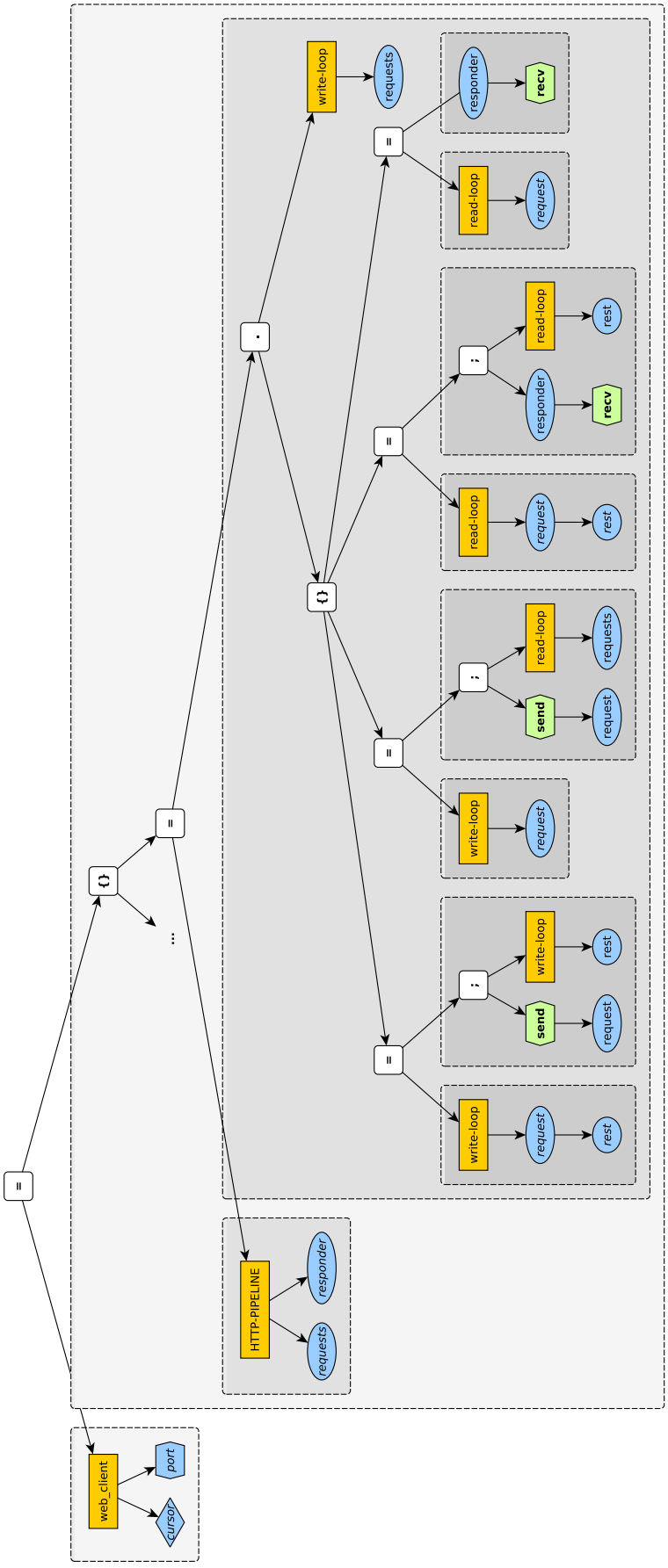

Figure 6: A pipelining web client