River: An Infrastructure for Context Dependent, Reactive Communication Primitives

Jong Hee Kang[1], Matthai Philipose[2], Gaetano Borriello[1,2]
Department of Computer Science and Engineering[1]
University of Washington, Seattle, WA
{jhkang,gaetano}@cs.washington.edu
Intel Research Seattle[2]
matthai@intel-research.net

ABSTRACT

Applications and services in ubiquitous computing systems often interact in a context-dependent, reactive manner. How information flows, and what services communicate when, is determined by the context of the physical space, the context of users, and the services that are available. Existing discovery systems provide basic facilities for finding services based on their static descriptions in the form of attributes. Context-dependent attributes are not included in the service advertisements as they may in turn be computed and stored in other services. We propose an infrastructure, called River, that provides various context-dependent, reactive communication primitives. These primitives are implemented using a single underlying technique called Relational Query Based Dispatching (RQD), which views the set of services in the system as a federation of databases, the discovery service as a distributed query processor for these databases, and communication as a combination of query processing and dispatching. In this paper, we describe the design and implementation of the River infrastructure. We also show that the context-dependent query processing within the discovery service can actually reduce the lookup latency with little effect on overall throughput.

1. INTRODUCTION

A programming model that has received much recent attention is dynamic service composition [1][2][3] [11][12]. In this model, a network environment is populated by a number of independent processes (called *services*). Each service typically depends on, and communicates with, others for its functionality. Dynamic composition burdens programmers with two key issues:

- **Context sensitivity.** Services tend to be specialized to be applicable in specific situations, or *contexts*. The set of services communicated with is therefore context dependent: it may depend in complex ways on data produced and stored throughout the system. The logic for specifying the appropriate services to link with is therefore substantially more complex than in traditional programs.
- **Reactivity.** Services representing devices, especially sensors and actuators, are common. Sensors add sensed data to the system constantly, leading to rapidly changing context. As the context changes, a service may need to use different actuators or services than before. Each service therefore needs logic that frequently rediscovers and rebinds to appropriate and available services.

Writing services that address these issues robustly and with high performance can be quite challenging, even for sophisticated programmers. In this paper, we describe the River system, which provides context dependent, reactive versions of four common communication primitives, using a common underlying technique called *Relational Query Based Dispatching (RQD).*

RQD incorporates two key ideas. First, in addition to offering traditional communication interfaces such as those using calls and events, every service is also allowed to advertise a set of schemas and relational operations such that they support the advertised operators over (a subset of) the relations specified by the schemas. The set of all services can then be viewed as a federated relational database, and complex relations within the data can be expressed as queries in standard relational languages such as SQL. In effect, a relational query processor replaces the traditional discovery service. Second, programmers may define the addresses of endpoints of communication primitives (specifically function calls, events, connections and relational queries) declaratively using relational queries over the federation. The system assumes responsibility for binding the endpoint to the most appropriate address at any given time, and for dispatching data to this address.

A few recent systems [1][2][3][5][8][10] have addressed the problem of context dependent reactive communication primitives. In most cases, the underlying solution is similar

to that provided by River. Each system provides a declarative language or notation to specify a communication end-point, and dynamically rebinds the endpoint to the appropriate concrete address when the specification evaluates to a new value. River's novelty is in the scope and expressiveness of its query language (in particular, most other systems only allow queries to match against advertisements, not arbitrary data contained within services; most proposed query languages are not as expressive as SQL), the variety of communication primitives it supports (other systems typically each support one or two primitives), and its demonstration that a single underlying framework can support all the primitives.

The rest of the paper is structured as follows. Section 2 presents motivating examples. Section 3 describes the design of River. Section 4 describes our implementation and preliminary evaluation. Section 5 compares the River approach to related ones. Section 6 concludes with an extensive discussion of future work.

2. MOTIVATING EXAMPLES

We describe, for each of the primitives we support, a scenario in which it would be used, how the scenario could be handled using a conventional discovery-service based approach, how the scenario is handled in River, and the possible benefits of the River approach.

We assume that in the conventional approach [4][9][11] the only support provided by the system (via the discovery service) is to map an attribute-based description of a service onto a reference to the service (which we call a *service ID* below). Once services discover target services, they manage for themselves all communication with the target services.

2.1 Function calls

Consider writing a function, that when invoked, turns on a lamp close to Alice. We assume that the location of both Alice and the lamps in the space is variable. We assume further that each lamp has a unique *device ID*, and is represented by a proxy service. We also assume several location services exist to provide location of an object given its ID. In principle, the code should check, for each lamp in the system, whether a location service maps its ID to Alice's location, and if so, ensure that lamp does turn on.

Figure 1(a) shows, in practice, how the body of such a function would be written in the conventional approach. The code must explicitly discover services (in this case lamp proxy and location services) of potential interest via lookups on the discovery service, flow information between services in the appropriate manner (in this case, use the lamp id to get its location, and then compare the location to Alice's), and iterate until it finds a service that is accessible and relevant (in this case, we need to find a location service that has information on the lamp, and a lamp that successfully turns on).

```
Set lamps= Discovery.lookup("device_type=lamp");
Set locs = Discovery.lookup("service=location");
foreach LocSvc loc in locs {
 if(alice_loc = loc.getLocation("Alice")){break;}}
foreach Lamp lamp in lamps {
  foreach LocSvc loc in locs {
    Loc lamp_loc = loc.getLocation(lamp.id);
    if (lamp_loc == alice_loc) {
      if (lamp.turnOn()){break; //early exit}}
}}
```

(a) Conventional approach

```
String query =
 "SELECT D.service_id
  FROM dev_svcid_tbl D, loc_tbl L1, loc_tbl L2
  WHERE D.device_type = 'lamp' AND
        D.device_id = L1.device_id AND
        L1.location = L2.location AND
        L2.id = 'Alice'";
LampCommand cmd = new LampCommand("turnOn");
…
RiverDiscovery.call(query, cmd);
```

(b) The River approach

Figure 1. Reactive, context-dependent function calls

Figure 1(b) shows how the same effect is achieved in River. Most of the work is in the definition of the SQL query named `query`. In defining this query, the programmer is able to view the discovery service as containing table `dev_svcid_tbl` with schema `dev_svcid_tbl (service_id, device_id, device_type)`, and all the location services in the system as a single table `loc_tbl(object_id, location)`. Given this view, the programmer defines the services of interest in terms of relational operators on these tables. He then packs the name and arguments of the function to be invoked into a command. Finally, he uses the `RiverDiscovery.call` function to associate the call and its arguments with the query, and to request execution of the call on one service that satisfies the query.

Two points are worth noting. First, the programming burden on the programmer of explicitly discovering, flowing information between, and iterating to find relevant services is substantially relieved. Second, by using the `RiverDiscovery.call` whenever he would use a statically bound remote procedure call, the programmer can ensure that the calls are reactive, in the sense that they will bind to the most appropriate target each time.

2.2 Events

Services, especially those representing sensors, often *publish* streams of data objects or events. Services interested in being notified of these events express their interest by *subscribing* to the subset they are interested in. In all cases we are aware of, a service specifies this subset as *dispatch constraints* on the value of the incoming event.

```
Discovery.subscribe("type=motion", new Handler());
…
class Handler{
void handle(MotionSensorEvent e){
 Set tmps = Discovery.lookup("service=temp_s");
 Set locs = Discovery.lookup("service=location");
 foreach TempSvc tmp in tmps {
  foreach LocSvc loc in locs {
   Loc l = loc.getLocation(e.device_id);
   if(tmp.getTempAtLocation(l) < 60){
L:  …          //process the event
    return;    //early exit from loop}
 }}
…}
```

(a) Conventional approach

```
String query =
  "SELECT T.location
   FROM temp_tbl T, loc_tbl L, event
   WHERE event.type = 'motion' AND
         T.temperature < '60' AND
         T.location = L.location
         L.id = event.device_id)";
RiverDiscovery.subscribeEvent(query,
                              new Handler());
class Handler{
void handle(MotionSensorEvent e){
… //same code as at L above
}
```

(b) The River approach

Figure 2. Reactive, context-dependent events

In many emerging applications [7], however, the relevance of an event is determined not just by the value of the event itself, but also by context data stored in other services in the system.

Consider a building automation application that automatically controls temperature. Suppose it needs to be notified of motion in parts of the house where the temperature is below some threshold, say 60^{O}F, so that it can selectively turn up the temperature. Say motion sensors all over the house constantly publish packets of data that contain their device ID and a boolean indicating the presence of motion, that temperature is provided by a number of services that map between location and temperature, and that location is provided by services mapping between object ID and location.

Figure 2(a) shows how the application would be written using a conventional event dispatch system. The application registers interest in all events from motion sensors, and registers a handler for these events. A crucial point is that because the dispatch constraint (here `"type=motion"`) is restricted to information contained in the incoming event, it is impossible for the programmer to be more specific about which motion sensors he is interested in. On receiving an event the handler uses the motion sensor device ID in the event to locate it, and uses the location to look up its ambient temperature in a temperature service. In addition to the burden of writing the code that robustly implements the logic as before, this approach has the intrinsic performance problem that it requires the application to handle (and filter) all motion sensor events under the possibility that they may be relevant.

Figure 2(b) shows how the application would be written in River. The dispatch constraint on the incoming packet is specified declaratively using a relational query; and the constraint may reference *any* database in the current federation. The River event dispatcher can therefore be more selective in the events forwarded to the application than the conventional dispatcher. A minor wrinkle is that the query refers to the special table named `event`. River binds this table name to the incoming event, so that we can access fields of the event using standard SQL notation.

2.3 Connections

Services processing streaming media often use persistent connections (such as sockets) to communicate large, continuous streams of data to other services. Traditionally, the endpoints of the connection remain fixed for the duration of its flow. In emerging applications, however, the endpoints may change during the flow.

Consider a baby monitoring application that uses cameras distributed around a house to deliver a video stream of a baby's current antics to the display closest to its mother. Because both baby and mother move around the house, the nearest display and camera respectively may change every few seconds, so that the application needs to reset the connection fairly often.

Figure 3(a) shows how such an application would be implemented using traditional means. The fragment of code is designed to execute in a separate thread from the main application. It essentially loops until the application exits, springing into action every `SAMPLING_INTERVAL` microseconds. On each iteration it discovers appropriate location, camera and display services, and if the required camera or display has changed, requests the camera service (as originator of the connection) to terminate its connection to the old display, create one to the new one, and start streaming data through the new display (all in the line labeled **M**). We have simplified the application for clarity: in practice the programmer would need to make provisions to minimize hysterisis, check for device failure.

In spite of the simplifications, the code is still quite complicated: both discovering appropriate services robustly, and achieving the connection reset are non-trivial. Also, the performance of this synchronous, polling based approach can be bad. A sophisticated programmer would want to consider the trade-off between polling at long-enough intervals and adding machinery for event-based updates (assuming the location service supported the required notifications).

```
Svc curCam = curDsp = null;
while(!exitApplication){
 Set cams = Discovery.lookup("service=camera");
 Set dsps = Discovery.lookup("service=display");
 LocSvc l = Discovery.lookup1("service=location");
 foreach CamSvc cam in cams {
  Loc camloc = l.getLocation(cam.id));
  foreach DspSvc dsp in dsps {
   Loc dsploc = l.getLocation(dsp.id);
   if(camloc == l.getLocation("baby") &&
      dsploc == l.getLocation("mom")){
    if(curCam == cam && curDsp == dsp){goto L;}
M:  cam.off(curDsp);cam.connect(dsp);cam.on(dsp);
    curCam = cam; curDsp = dsp;
    goto L;}
 }}
L: sleep(SAMPLING_INTERVAL);}
```

(a) Conventional approach

```
String camQuery =
  "SELECT D.service_id
   FROM dev_t D, loc_t L1, L2
   WHERE D.device_type = 'camera' AND
         D.device_id = L1.id AND
         L1.location = L2.location AND
         L1.id = 'baby'";
String dspQuery =
  "SELECT D.service_id
   FROM dev_t D, loc_t L1, L2
   WHERE D.device_type = 'display' AND
         D.device_id = L1.id AND
         L1.location = L2.location AND
         L1.id = 'mom'";
Connection c = RiverDiscovery.connect(camQuery,
        dspQuery, new ConnectionStatusHandler());
```

(b) The River approach

Figure 3. Reactive, context-dependent connections

Figure 3(b) shows how the same application would be implemented in River. As usual, endpoints of the primitive are specified in a clean, declarative manner. The machinery for resetting connections is handled by River, inside the `connect()` function call. In addition to the usual two queries denoting the two endpoints of the connection, note that the call has a connection status handler argument. The connection status handler is invoked by River when there is any change in the connection status.

By providing code for robust and high-performance implementation of reactive connections, River clearly reduces the burden on the programmer. Further, since the one-time cost of adding a sophisticated optimization to this code is amortized over many users, River can potentially provide a high-performance implementation.

2.4 Relational Queries

A happy side effect of being able to consider the set of all services as a federation of databases is that programmers can issue arbitrary SQL queries over the federated database. In the previous three sections, these queries were devoted to deciding the control flow of traditional communication primitives. However, the `RiverDiscover.lookup()` method is essentially evaluates an arbitrary SQL query over the federated database, and is directly available to the programmer. In many cases, using such a query directly as communication primitive may be the most natural option for a programmer.

3. DESIGN

The River discovery service consists of the *query processor* and the *dispatcher* (Figure 4). The query processor enables context-dependent naming of services by allowing the client applications to specify the name of the services with which they intend to interact using context-dependent information provided by other services in the system. The dispatcher integrates the communication primitives with the context-dependent naming and makes them reactive to context changes.

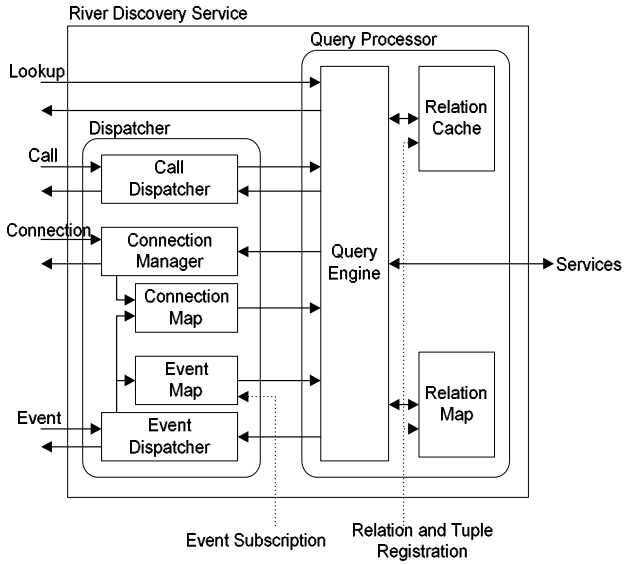

Figure 4. The structure of the River discovery service.

River views the set of services in the system as a federation of databases, and each service serves the information it maintains in the form of a database relation. The query processor processes queries written in SQL over these databases. The query processor consists of a *relation map* that maintains the information about which service has which relation, a *relation cache* that stores actual data tuples locally for better performance, and the *query engine* that parses and processes the query.

The dispatcher consists of the *call dispatcher*, the *event dispatcher*, and the *connection manager*. The call dispatcher receives a late-binding function call request from an application, determines a destination for the function call using the query processor, and forwards the call request to the destination. The event dispatcher receives an event from an event source, determines the event subscriptions matching the event, and sends the event to the subscribers. All the event subscriptions are stored in the event map. The connection manager manages the connections registered in

the discovery service. It re-evaluates the query for each end-point of each connection periodically or upon events, and changes the connection end-points when necessary.

```
interface RiverDiscovery {
  Handle registerRelations(String[] names);
  Handle registerTuples(String name, Set tuples);
  Handle subscribeEvent(String query,
                        EventHandler eh);
  void withdraw(Handle h);

  Set lookup(String query);
  Set call(String query, Message m,
           boolean multicast = false);
  void asyncCall(String query, Message m,
                 boolean multicast,
                 ContinuationHandler ch);
  void publishEvent(Event e);
  Connection connect(String q1, String q2,
                     ConnectionHandler h);
}
```

Figure 5. Interface of the River discovery service stub

Figure 5 shows the interface of the River discovery service stub. The first four methods are used for registration and deregistration, and the last five methods are for communications.

3.1 Registration

When a service is started, it registers the names of the database relations it maintains with the discovery service. For example, a location service that has a mapping between an ID and the current location of the person or object of the ID may register the relation name '`loc_tbl`' whose schema is '`(object_id, location)`'. We assume that the name of a table is unique (this can be enforced by yet another service so that databases can be easily aliases between environments), and the schema of a table is available using that name. The relation names and the services maintaining the relations are stored in the relation map. The relation map is a table with the service ID (that includes the address of the service) and the relation name fields. For a given relation name, the relation map returns the service IDs of all the services that have the relation.

In addition to the relation names, a service can also register actual data tuples along with the relation names. Usually, services register data tuples when the number of data tuples in the relation is rather small and the contents of the relation are static and do not change frequently. The data tuples are stored in the relation cache.

Caching data tuples inside the relation cache emulates conventional discovery services. In conventional discovery systems, a service registers its static attributes with the discovery service. Likewise, in River, a service can register the attributes composed in a tuple with the River discovery service. For example, a device proxy service has several attributes that describe the service and do not change: the service ID, the device ID of the device it represents, and the device type. The device proxy service can register the '`dev_svcid_tbl`' relation whose schema is `(service_id, device_id, device_type)` and a tuple for the '`dev_svcid_tbl`' relation that has the attributes for this proxy service.

Applications or services interested in certain events register their interests with the discovery service. The event registration has two parameters: the interest for the event written in SQL and the event handler for the event that is to be invoked when the event is dispatched. The event registration is sent to the event map that maintains a table with the event subscription query, the event handler ID, and the service ID of the subscriber. When the event map receives an event registration, it parses the event query before storing it into the table to reduce the event dispatching time.

3.2 Query Processing

The query engine receives lookup requests directly from client applications or from the dispatcher. The query engine has two `lookup` methods (Figure 6). The lookup request from client applications and call dispatcher uses the first lookup method with a query string as a parameter. Upon receiving the query, the query engine parses the query and then makes a query execution plan. When generating query execution plans, the query engine first checks the relation cache and then refers to the relation map to figure out the location of each relation. The query execution plan includes details such as the sequence of operations, and how the queried relations are accessed (whether from the relation cache or from other services). Once the query execution plan has been generated, the query engine executes the query and returns the result.

The second `lookup` method takes a parsed query and an event as parameters and is used by the event dispatcher and the connection manager. They have parsed queries stored in the event map and the connection map, respectively, and send a parsed query and an event (if the call is triggered by an incoming event, otherwise null) as parameters to the query engine for lookup. Because the query involves the incoming event in this case, as an optimization, the query engine first evaluates the query with the event to determine whether the query requires further evaluation. If the query turns out to return null after evaluating with the event, the query engine returns null immediately without generating and executing an execution plan. If the return value is non-null, then the query engine generates the execution plan and executes it.

```
interface QueryEngine {
  Query parse(String query);
  Set lookup(String query);
  Set lookup(Query parsedQuery, Event e);
}
```

Figure 6. Interface of the query engine.

3.3 Dispatching

3.3.1 Dispatching Function Calls

The River discovery service provides two late-binding function call methods (Figure 5), and the call dispatcher processes them both. The first method – `call` – is a synchronous function call method. It takes a query string, a function invocation message, and a multicast option as parameters. To invoke the call method, the caller composes a query for finding the services to invoke and the invocation message that includes the invoked service's method name and parameters. When the discovery service stub's `call` method is called, the stub adds the caller's address into the invocation message and sends the query string, the modified invocation message, and the multicast option to the call dispatcher of the discovery service. The call dispatcher's `call` method takes this request and invokes the `lookup` method of the query engine with the query and finds the services to invoke that match the query. If the multicast option is set to true, it sends the function invocation message to all the possible services it finds. If the multicast option is set to false, it sends the invocation message only to the first matching service among them. Then, it notifies the stub of the number of services invoked. The stub waits for responses from all the services invoked. The invoked service returns the result back to the caller whose address is included in the invocation message. When the stub receives results from all the invoked services, it packs the results into a set and returns it to the caller.

In addition to the synchronous function call method, River also provides an asynchronous function call method `asyncCall`. It takes a continuation handler as well as a query string, a function invocation message, and a multicast option as parameters. When the `asyncCall` is called, the discovery service stub adds the caller's address and the continuation handler ID to the invocation message and invokes the call dispatcher's `call` method. Then, the stub returns immediately and the caller can continue its execution. Every time an invoked service returns its result, the continuation handler is invoked. Thus, if the multicast option is set to true, the continuation handler may be invoked more than once if more than one matching services are found by the query engine.

```
int call(String query, Message m, Boolean mc) {
  Set s = QueryEngine.lookup(query);
  int servicesToInvoke = 0;
  foreach ServiceID sid in s {
    send(sid, m);
    servicesToInvoke++;
    if (!mc && servicesToInvoke == 1) {return 1;}
  }
  return servicesToInvoke;
}
```

Figure 7. The call method of the call dispatcher.

3.3.2 Dispatching Events

A service that wants to publish an event uses the `publishEvent` method. The event is passed as a parameter to the event dispatcher of the discovery service. When the event dispatcher receives an event, it iterates through all the registered event subscriptions in the event map and calls the lookup method of the query engine for each subscription. If the query engine returns null, it means that the incoming event does not match with the query. If the query returns a result that is not null, the event dispatcher attaches the result to the event and sends the event to the subscriber with the event handler ID. The subscriber's address and the event handler ID are stored in the event map table with the query. Note that the event dispatcher attaches the query result to the event when dispatching it to the matching subscribers. Thus, the subscribers can receive some additional information in addition to the event itself. For example, the subscriber may receive a motion event to which the location of the event is attached (in the example in section 2.2).

```
void publishEvent(Event e) {
  foreach Subscription s in EventMap {
    Query q = s.query;
    if((Set r = QueryEngine.lookup(q, e))!=null){
      e.attachData(r);
      e.attachHandlerID(s.handlerID);
      send(s.address, e2);
    }
  }
}
```

Figure 8. The publishEvent method of the event dispatcher

3.3.3 Managing Dynamic Streaming Connections

A reactive streaming connection can be established with the `connect` method. It takes two query strings for two end-points and a connection handler for getting connection status information. The discovery service stub sends the connect request to the connection manager of the discovery service. The connection manager's connect method takes the request and establishes a connection between the two endpoints. It first parses the two query strings and finds corresponding services for the two endpoints. If services are found, it asks the first endpoint to make a connection to the other endpoint. Then, it stores the connection information including parsed queries into the connection map. Finally, it returns the connection handle to be used by the application to control the connection.

The connections stored in the connection map need to be reevaluated so that they can react to context changes that require changing connection. The reevaluate method is invoked periodically by the connection manager, or invoked when an event comes to the event dispatcher. This method reevaluates the queries and changes the connection if either of the endpoints has changed. Also, it notifies the application of any change in the connection status.

```
Connection connect(String q1, String q2,
                   ConnectionHandler ch) {
  Query parsedQ1 = QueryEngine.parse(q1);
  Query parsedQ2 = QueryEngine.parse(q2);
  Set s1 = QueryEngine.lookup(parsedQ1, null);
  Set s2 = QueryEngine.lookup(parsedQ2, null);
  if (s1.empty() || s2.empty()) return null;
  ServiceID sid1 = s1.getAt(0);
  ServiceID sid2 = s2.getAt(0);
  ConnectionID h = sid1.connect(sid2);
  Connection c = ConnectionMap.store(ch,
           parsedQ1, parsedQ2, sid1, sid2, h);
  return c;
}

void reevaluate(Event e) {
  foreach Connection c in ConnectionMap {
    Set s1 = QueryEngine.lookup(parsedQ1, e);
    Set s2 = QueryEngine.lookup(parsedQ2, e);
    if(s1.empty() || s2.empty()) {
      c.ch.notify("lost"); continue; }
    if(s1.has(c.sid1) && s2.has(c.sid2)) continue;
    ServiceID sid1 = c.sid1, sid2 = c.sid2;
    if(!s1.has(c.sid1)) sid1 = s1.getAt(0);
    if(!s2.has(c.sid2)) sid2 = s2.getAt(0);
    sid1.disconnect(c.h);
    ConnectionID h = sid1.connect(sid2);
    c.ch.notify("rebound");
    ConnectionMap.update(c, sid1, sid2, h);
  }
}
```

Figure 9. The connect and reevaluate methods of the connection manager.

4. IMPLEMENTATION AND EVALUATION

4.1 Implementation

The River framework is implemented on top of Rain [9], an asynchronous event-based service/messaging system. Messages in Rain are in XML and the format of the message can be easily changed. The only required fields are sender and recipient tags. The River discovery service and all the other services are implemented as Rain services. Rain has been implemented in several different languages, and we use the Java version of Rain for our implementation.

Most of the major components of the discovery service have been implemented except for the connection manager which is still under development and only partially functional.

4.2 Performance Evaluation

We have measured the performance of the River discovery service. The experiments were performed on seven lightly loaded Pentium 4 (3.0 GHz, 1 GB RAM) machines. The discovery service runs on one machine, all other participating services run on 5 machines, and the last machine is used for measuring.

The most important aspect of the River discovery service that influences its performance is the query processor. In River, all the query processing operations including accessing remote services are done inside the discovery service. In conventional discovery systems, however, the discovery service only performs queries on data stored within the discovery service itself and the application needs to access the remote services explicitly. Thus, the basic question we set out to answer was how the River-style context-dependent query processing performance compares with the application-explicit version of query processing in conventional discovery systems.

First, we measured the latency of the lookup operation. The lookup operation is needed for an application to find a service for a given query. In River, the latency is the time that transpires while the application is waiting to get this result. In the conventional case, the latency includes the time for the application to get partial results from the discovery service and the time to contact remote services for further processing of the query. Figure 10 shows the latencies of the River version and the conventional version. When there is no remote relation involved in the query, in other words, the query is focused on the service attributes already stored in the discovery service, there is no noticeable difference between the two cases. However, as the number of remote relations involved in the query increases, the latency of the conventional case gets longer than the River case. That is because the conventional case needs one more round-trip to the discovery service for each remote relation it has to execute as part of its query. In the River case, the remote relation access happens in the discovery service itself, and the address of the service can be found locally, and thus, more efficiently, within the discovery service itself.

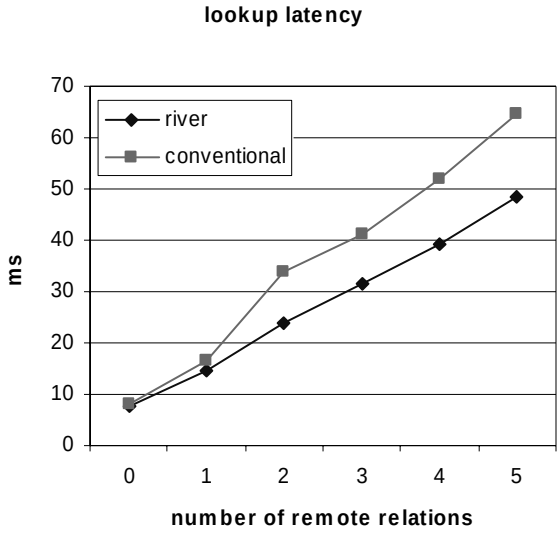

Figure 10. Lookup latency of the River case and the conventional case.

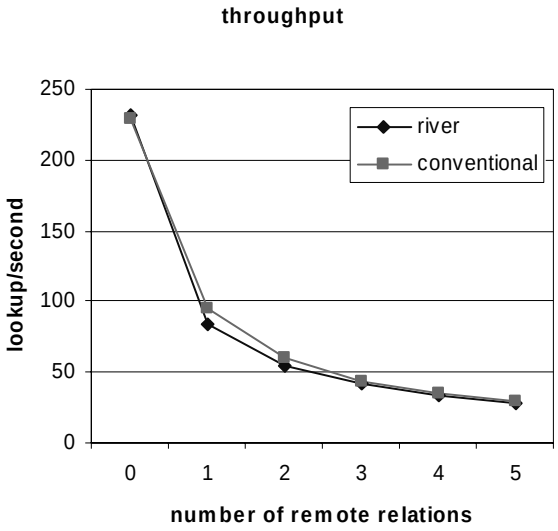

Figure 11. Throughput of the River case and the conventional case.

Next, we measured the throughput of the River discovery service. In the River case, all the query processing is performed by the discovery service, and the discovery service consumes more computing power than in the conventional case where much of the query processing is done at the application. As expected, the throughput of the conventional discovery service is higher than that of the River discovery service. However, even though the query processing is done in the application in the conventional case, the application still needs to ask the discovery service of the addresses of the services with remote relations and it consumes the resources of the discovery service. This communication cost of the discovery service (receiving request and sending reply) offsets the computation cost (processing query). That is why the throughput difference is negligible.

The measurement shows that the River-style context-dependent query processing inside the discovery service can give much better latency with little sacrifice in throughput.

5. RELATED WORK

There has been a flurry of recent work towards supporting context dependent and efficient discovery between services, and (to a lesser extent) on ensuring that communication between services is reactive. We compare River with these applications along three dimensions: the operations supported by the query language used to specify the attributes required of the target service, the variety of external attributes (i.e. those not explicitly stored in the discovery service) that can be reference in the query, and the variety of reactive control primitives provided.

Table 1 summarizes the comparison. The first column of the table names the system, the second through fourth handle each of the above three dimensions, and the last mentions other important related features of each system. Below, we discuss each column of the table in turn.

We begin with the query language (column 2 of the table).

Table 1. Comparison of the related works.

System	Query Language	External Attributes In Query	Reactive Communication primitives			Other Features
			Function Call	Event	Connection	
Jini [11]	java interface with attributes-value pairs	none	no	no	No	commercial system
SDS [4]	attribute-value pairs	none	no	no	No	
INS [1]	hierarchical attribute-value pairs	none	yes	no	No	highly scalable, allows frequent update of advertisements
SoNS [10]	non-recursive lisp-like expressions	none	no	no	Yes	socket-level support
EventHeap[5] TSpaces [8]	attribute-value pairs (templates)	none	no	yes	No	centralized, all communication modeled as database reads and writes
Solar [2]	path-structured name	location only	no	yes	No	can name and share streams output from composed services
iQL [3]	iQL	values produced by other services	no	yes	No	fully reactive language
River	SQL	values contained in advertising services	yes	yes	Yes	system from this paper

A common format for specifying matches is as attribute value pairs. SDS, EventHeap and TSpaces support straight attribute-value pair matches. Jini, allows specification of the Java interface of the target services in addition to attributes. INS allows attributes and values to be placed in hierarchies. Solar specifies services explicitly by name, and provides a path-based formalism for naming the result of composing services. In all these cases, service identification is a matter of structurally matching queries to values. A smaller set of systems supports *interpretable* queries. SoNS allows queries that are expressions (with relational and arithmetic operators) over attribute-value pairs. iQL supports expressions in a general reactive language. Finally, River supports expressions in SQL.

As our earlier examples show, the more expressive the query language, the less the user has to do in evaluating the query. In the extreme case, of course, the queries could be fully general functions. However, this may result in queries that do not terminate and are difficult to optimize.

We now move to the scope of the query (column 3 of the table). As we have argued, an important aspect of the support for context dependent control in practice is the ability to consult arbitrary services in the system as part of the endpoint-binding query. Most systems do not allow queries to make such references. The exception is iQL and Solar. iQL allows services to reference values published by other services (iQL has a pure event-filter-based model). Solar's naming conventions location context in the name space. River, as explained previously, allows queries to reference and operate on data stored in any service that has advertised its database schema.

Finally, consider the provision of reactive communication primitives (column 4). An extreme in this category are the pure discovery services such as Jini and SDS, which provide no support at all for communication, beyond returning relevant target services. INS provides a late-binding function call that has semantics similar to that of River function calls. Service-oriented Network Sockets (SoNS) integrates the discovery lookup into the socket API with semantics similar to River connections (except that they support migration of only one endpoint). Finally, a number of systems support reactive events: in some sense, since traditional events already work by matching query patterns from subscriptions to published data, they already have "most relevant service" semantics. iQL in particular is noteworthy in that it provides expression level reactivity: not only does the appropriate event get forwarded, even expressions get re-evaluated and re-bound, when inputs change.

River is able to provide all three communication primitives using a single programming metaphor of query-defined endpoints, and the implementation technique of relational query based dispatching. We expect that allowing the programmer to use their primitive of choice (while not worrying about low-level details) will improve their productivity.

A final category of related work not included in the table is that of distributed relational databases. A very large body of work exists on this topic. The survey by Kossmann [6] is a good starting point. Although we intend to use techniques from the literature to optimize our system as needed, there are some significant differences between our applications and distributed databases in general. First, we expect many of our services to run on resource-impoverished devices. Second, we expect many services (and therefore databases) to enter and leave in the order of minutes and hours. Third, we expect that the pattern of queries seen by the River query processor will have much higher regularity than typical database queries (since they will be implementing the same control primitives repeatedly). Thus, it is an open question to what extent techniques applied successfully will transfer over to River.

6. SUMMARY AND FUTURE WORK

We have argued that many emerging applications entail a combination of "deep context dependence" and reactive communication primitives, both of which are supported only in limited ways in recently proposed systems. We have presented a novel technique, relational query based dispatch (RQD), that provides a single unifying framework within which conventional communication primitives such as function calls, events and connections can be made both context dependent and reactive. Our preliminary measurements show that the high level of abstraction we provide to the programmer can potentially be supported at little performance cost. In some cases, communication with RQD can even be faster than without.

Although early results seem promising, we anticipate substantial further work to validate and improve the basic RQD model. One of the biggest concerns is that although we provide an attractive abstraction to the clients of various services, we impose a fairly heavyweight model (that of a federated database) on the system as a whole, thus potentially paying an unacceptable cost in resource usage and performance. Although the measurements presented above are encouraging, we fully expect scenarios where the federated database, if naively implemented, generates large amounts of extra traffic and computation. Our first order of business, once we complete the implementation of the connection manager, will be to do a much more extensive study of the performance limits and bottlenecks of River.

As mentioned in section 5, much work has already been done on optimizing query processing in (federated) databases. We will use the results of our performance study to select the optimizations that best alleviate our performance bottlenecks while not imposing unreasonable

requirements on our constituent services. Given our early understanding of the system, we expect certain techniques to be quite valuable. Since RQD results in the same query being re-executed multiple times, we expect caching of query results to be useful. In order to invalidate cached data, we expect asynchronous notification of change in query results to be useful. In order to avoid transferring large quantities of data across the network, we expect locality-aware query plans to be useful. In order to support applications in the absence of powerful centralized servers, we are considering a distributed peer-to-peer style implementation of the central dispatcher and query-processor. In order to support resource limited devices, we are exploring principled techniques for providing "limited" relational services.

A final direction we are investigating is tighter integration with the programming language. River is currently implemented as a library in Java. Although a library-based implementation is simpler than augmenting the language, it increases the possibility of statically undetected errors in the program, and potentially sacrifices optimization opportunities. Since all our relational queries are currently implemented as strings, for instance, it is entirely possible that the queries may not even parse correctly, let alone return values of expected type when run.

We currently use River as the infrastructure for implementing a demonstration application targeted at home-based eldercare. We are collaborating on porting a more deployment-ready version of this application on to River. Once River is stable, and performs reasonably, we hope to test it on a wider variety of applications that require dynamic service composition.

7. REFERENCES

[1] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. *The design and implementation of an intentional naming system*, Proc. 17th ACM SOSP, Kiawah Island, SC, Dec. 1999.

[2] Guanling Chen and David Kotz. *Context Aggregation and Dissemination in Ubiquitous Computing Systems*, Proc. 4th IEEE WMCSA, 2002.

[3] Norman H. Cohen, Hui Lei, Paul Castro, John S. Davis II, and Apratim Purakayastha. *Composing Pervasive Data Using iQL*, Proc. 4th IEEE WMCSA, 2002.

[4] Steven E. Czerwinski, Ben Y. Zhao, Todd D. Hodes, Anthony D. Joseph, and Randy H. Katz. *An Architecture for a Secure Service Discovery Service*, Proc. ACM/IEEE MOBICOM, August 1999.

[5] Brad Johnson and Armando Fox. *The Event Heap: A Coordination Infrastructure for Interactive Workspaces*. Proc. 4th IEEE WMCSA, 2002.

[6] Donald Kossmann. *The State of the Art in Distributed Query Processing*. ACM Computing Surveys, 32(4):422-469,2000.

[7] Anthony LaMarca, David Koizumi, Matthew Lease, Stefan Sigurdsson, Gaetano Borriello, Waylon Brunette, Kevin Sikorski, Dieter Fox. PlantCare: An Investigation in Practical Ubiquitous Systems. Proc. 4th UbiComp, 2002.

[8] T. Lehman, S. McLaughry, P. Wyckoff. *TSpaces: The Next Wave*. Hawaii Intl. Conf. on System Sciences (HICSS-32), Jan. 1999

[9] Rain. http://seattleweb.intel-research.net/projects/rain/.

[10] Umar Saif and Justin Mazzola Paluska. *Service-oriented Network Sockets*, Proc. MobiSys, May 2003. http://www.acm.org/sigs/pubs/proceed/template.html.

[11] Sun Microsystems Corporation. The Jini Architecture Specification, Ver 1.2, Palo Alto, California, December 2001.

[12] Universal Plug and Play. http://www.upnp.org.