Architectural Models for Visual Sensor Networks

Zhenyuan Zhao
Department of Computer Science
Brown University

1. Introduction

Visual sensor networks (VSNs) are networks composed of very large numbers of visual sensors (smart cameras). Different from today's simpler sensor networks which tend to be aggregators or monitors of low-bandwidth data, VSNs are able to capture still images, short bursts of video frames and continuous video streams, thus face new challenges due to their high data rates. The bandwidth required for all the visual sensors to transmit their video data to a central location for processing or storage would be enormous. As a result, new infrastructure needs to be introduced. One solution to this problem is to push high-rate data processing (such as image processing) toward the sensors, so that only produced lower-rate data (such as object features) needs to be transmitted. This project aims to solve these problems.

To be more specific, the project involves the design and implementation of

- Novel real-time image processing algorithms that take advantage of specialized hardware. Those algorithms will be able to work in a distributed fashion so that multiple smart cameras can collaborate. In fact, it is very important to maximize the data processing and decimation at the sensor level and to minimize the amount of data transmitted.
- A data-flow oriented software platform to simplify the construction of VSNs. Ideally, a collection of image processing primitives will be developed, which allows programmers to specify behaviors of the data flow using block diagrams while isolating them from details about configuration parameters, like location, bandwidth, connectivity and so on.
- Scalable low-power routing protocols that function among disparate radio technologies.

This project will also be focusing on issues arising from dynamic wireless VSNs, for example, dynamic calibration. For dynamic wireless VSNs, sensors' locations are not known in advance, nor can we easily calibrate them. It would be very valuable if one sensor can figure out the relative location with respect to other sensors by asking data from them.

The report is organized as follows. Section 2 introduces previous attempts. Section 3 describes major components implemented to build current systems which will be introduced in Section 4. Section 5 discusses the future work.

2. Previous Work

The project studied multiple feature detectors and matching algorithms. In order to do that, 6 pairs of USB cameras were set up in the multi-media lab in B&H Building. Each pair of cameras was finely calibrated and synchronized (it was controlled by a laptop, so that two cameras took pictures at the same time). We took many high resolution pictures and processed them in Matlab. We used Harris corner detector to find the locations of features first, and then used SIFT feature descriptor to describe them and do the matching, which gives us more accuracy when computing their 3D locations. Also, our early results showed that color histogram did reasonable job in matching objects. However, SIFT feature did not in our case. Even for objects in two consecutive frames of the same camera, we were only able to find very few matches. We have tried changing parameters of the algorithm, but improvement was very limited. Also, SIFT feature detection and matching were a lot slower when applied on those high resolution images. Later on, we decided to build a working real-time system, so we moved to C++. Libraries that we adopted to process images are OpenCV and VXL.

We met some problems when developing the real-time system with the USB camera setup. First, due to USB connection, we were not able to connect to cameras from over 3 hops away. That means we have to control cameras near where they locate, which introduces many inconvenience when recording video streams and running the system using real-time data. Second, USB cameras are not "fast" enough for real-time video streams. It has low frame per second rate, and even for low resolution images, it takes comparatively long time

for USB connection to transmit. Moreover, since SIFT feature does not perform well, we do not have to stick to high resolution images. So we set up some network cameras in the CS Department. They are easily accessible via network connection and are capable of taking pictures of VGA size up to 10 frames per second. Network cameras do have limitations and will be replaced finally (which will be discussed in future work), but they work for current demos.

Two applications were developed using network cameras. One developed by Jie Mao and Mert Akdere is a set of libraries (bvsn library) with a tool that detects, matches and tracks objects within a single camera. Functions like building ground truth and analyzing blobs/objects relations were added to the tool in its later versions. Image processing methods adopted in the tool include background subtraction using Gaussian mixture model, color histogram, SIFT feature and spatial matching. Another application was developed by Cetin Koca. It detects objects and matches them using sort of clustering algorithm on color histogram. Additionally, it also provides a GUI developed in Qt which controls the cameras and gives a centralized view of approximate locations of the detected objects on a floor map. I have been adding new features to existing applications, such as displaying object trajectory, improving accuracy of camera view to floor map mapping, developing new functions in the library and so on.

There are two major limitations in both of the applications which do not fit into out ultimate goal. First, both applications process images in a centralized way. It slows down the node which does the processing when more cameras join the network. A more severe problem is the enormous bandwidth required to transmit raw images to the processing node. Second, all data is stored in a centralized data structure called bvsn_objectdb. So, that data structure is also likely to be the hot spot with high contention. We also need to make storage distributed. Otherwise, memory size limitation is a major concern. Because of the two limitations, scalability is not reached.

3. Building Blocks

My major work is to overcome those limitations and implement a working distributed system. To make things happen in a distributed fashion, new components have to be built, such as network, in-memory database and processing unit. Also, logic behind the GUI is modified to handle new models. Following section describes those major components.

Network

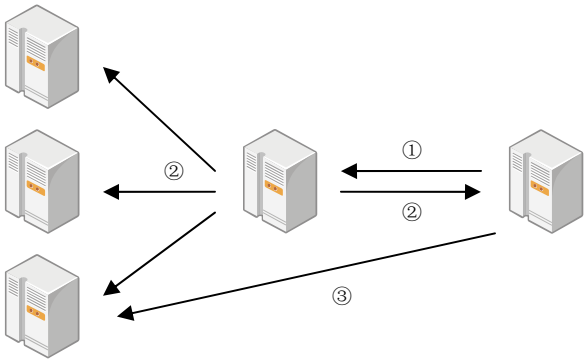
We cannot talk about distributed system without network. In fact, an efficient network layer is essential to our system. To meet our basic requirements, the network component needs to be able to send/receive anything we want, including raw frames, blobs, objects, various features and requests/responses. In addition to that, we need some infrastructure to handle requests received from socket, dispatch them to the right handler and send back the response once ready.

A set of packets are designed to hold data we want to send. I defined bnet_packet to be the base class that holds metadata of a packet only. All other packets implemented are derived from that class. For example, bnet_request_packet is the packet used to hold requests. It contains request type and a vector of arguments for that request (can be considered as calling a remote procedure). The request packet is used whenever some data needs to be fetched from other sensors. I adopt Boost to serialize those packets (thanks to Mao's suggestion). Intrusive way is introduced to serialize the packet itself. But for data structures defined in VXL library, non intrusive way is used.

Currently, our sensors are all wired cameras set up on TCP/IP network, and Boost Asio library is used to handle network issues. So the following discussion will be focusing on socket, rather than generic network. However, functions are designed to be loosely coupled, thus only small portion of code needs to be modified to use other kind of connection facilities, such as wireless. To achieve that, bnet_util is developed to send and receive packets. Also, bnet_camera_server is introduced to handle packets processing and other network issues, like establishing connections and dispatching inbound requests. bnet_camera_server also maintains a list of known sensors, which are the sensors that can be asked for data.

The network is set up like this. When a server is initializing, it adds itself to the list of known sensors and starts listening at user specified port. If the server is the first server to join the network, it does nothing. Otherwise, it needs to get servers list from other servers. We safely assume that before a server joins the

network, it already knows at least one server (named the known server). So the joining server will announce itself by sending all its known servers to the known server. In fact, at that time, its known servers list contains only itself. The known server, when receiving the announcement packet, will add the new server to its servers list, and send that list to all servers in that list. This step ensures any server in the network has the most up-to-date servers list. Figure 1 shows that process.



① New server adds itself to the know servers list, and joins the network by announcing itself.

② The known server, when receiving the announcement packet from some new server, will update its known servers list and then send the new list to all servers in that list.

③ Then the new server is able to ask some camera for data.

Figure 1. Network set up.

When the server is listening at some port, if an announcement packet is received, the aforesaid process applies. If a request packet is received, it will be dispatched to the right handler according to request type (defined in bnet_request_packet), followed by the response packet sent back. For example, if the request type is “get last frame of sensor x”, the server will check if it is “x”. If so, the local object database is asked for the frame, and he frame is sent back once fetched from the database.

In-Memory Database

We were using a data structure to hold all data of a camera in the previous system. However, there are other two limitations besides the centralized storage. First, data is not stored in a formalized way, so that we are not able to easily support formalized queries (for example, SQL). That means for each type of queries, we have to use different methods to traverse the data structure, match against the criteria and get results back, which will involve lots of work when later on, we decide to support more sophisticated queries. Another concern is search efficiency. For most cases, sequential searches are applied. It is easy, but slow. Though we can adopt a hash-like mechanism to do fast search and matching, the problem is whether it is worthwhile to do so. Again, we will meet the same problem, to develop different hash tables for different types of criteria. Also, it looks like is building indices on difference query criteria, so why not just use indices of a database? With those considerations in mind, we moved to in-memory database which fits very well.

An in-memory database is a real database that supports the ACID property. It differs from a traditional database because it primarily relies on main memory for data storage. As a result, it is much faster without disk I/Os introduced. This is very important to us because of fast query execution we expect. In fact, we can also build various indices and views to accelerate searches. I have tried three in-memory databases to be the backend of our bvsn_objectdb:

- MySQL: MySQL is able to create hash-based tables purely located in memory. It has C++ API, but we are only able to use it in a server-client mode. We want data to be in memory so that no disk I/Os will be introduced. However, if extra network I/Os need to be paid, we are still not gaining much.
- H2: H2 is the state-of-art in-memory database with very attractive performance. Again, we do not want to use server-client mode (H2 supports JDBC connection and is compatible with PostgreSQL’s protocol, experimentally, according to its documentation). H2 does support embedded mode, but is limited to Java only, since H2 is written in Java.
- SQLite: SQLite is an embedded SQL database engine. But unlike most other SQL databases, SQLite does not have a separate server process. Instead, it reads and writes directly to ordinary disk files (or in-memory) like a system call to file system. It is written in C, so that we can embed it into our code easily just like using a class. Drawback of SQLite is that it is not that feature rich, for example, it does not support foreign key, nor nested transactions. But we are able to avoid using those lacking features.

Finally, SQLite is adopted.

A very thin layer is built on top of SQLite, which is considered to be SQLite driver. SQLite does have a simple interface to run a bunch of SQL, but that interface cannot be used to insert or fetch a BLOB (a type defined in SQLite that can be any data in memory), which is required for frame storage. So a wrapper is provided to do that. Also, the original query model used in SQLite is the callback model. User submits a query, meanwhile providing a callback function, which will be invoked each time a row is fetched. Such model has its advantage, for example, it makes it much easier if complicated logic needs to be applied on the result. But in our case, the result is mostly very simple, and we do not want to provide or let users write many very simple callback functions. After looking at pqxx (PostgreSQL C++ API), I added a result type to SQLite. The result will be returned by submitting a query (empty result for some SQL, like insertions). If we consider the result to be a 2-dimension table, each cell of the table is referenced using a row number and a column name, so that users can access any part of the result easily. Another important function added is to support dumping the whole database, including schema and data, in SQL format. With that function, we can initialize a database using the dumped data, as well as loading a common database for debugging and analyzing purpose.

The new bvsn_objectdb provides interfaces to insert frames, blobs and objects, as well as some simple queries to retrieve data that meet some criteria, such as “get objects found in last processed frame”. The schema (see Appendix A) is derived from the E-R model of the classes in bvsn library. Taking the bvsn_blob class for example, in order to store a blob, its metadata is stored into the blob table, including where and when it is found, its location in the frame, width, height and size. If the blob is already processed with some features set, those features will also be stored in corresponding tables. For color histogram, its size, dimension and type will be stored in the blob_colorhist_desc table, and its values in a separate table. Two types of interfaces are provided to retrieve a blob from the database, the “shallow” version and the “deep” version. The shallow version only retrieves metadata of a blob. Most of the time, those information is enough for displaying purpose. But if matching needs to be done over two blobs to see if they belong to the same objects, features must also be fetched from the database. That’s what the deep version does.

Currently, data stored is relatively simple and straightforward. With no doubt, we need to support more queries and have a stronger query model (for example, component that turns queries into execution plans), thus more data have to be stored, mainly descriptive data. Also, there is no view built at present. If there are some frequent queries, it would be better to build views for them.

Processing Unit

Processing Unit is where images are processed. Given a raw frame, it first does background subtraction using Gaussian mixture model. After that, blob finding algorithm from VXL library is applied. We define a blob as a consecutive region in the boolean image produced by background subtraction. Once a blob is found, its properties never change. An object is defined as a collection of blobs, which are considered to be instances of the object at certain time. It is possible that an object consists of more than one blob even in a single frame, because there might be a case that an object appears in two frames captured different camera at the same time. So in our implementation, the bvsn_object class contains a map. Key of the map is the timestamp and value related to that key is a list of blobs found at that time that belong to the object.

The second step is to compute various features of each blob and to find an existing candidate object based on feature matching. Currently, the only feature used is color histogram. In fact, the best matching feature we found within a single camera is based on spatial information. That is, if two blobs are found in consecutive frames with about the same size and acceptable distance, they are very likely to be instances of the same object. However, this method does not work between multiple cameras, because blob location is a local feature as view of one camera can be very different from that of another. So the spatial information doesn’t make that sense for multi cameras. When matching a new blob, both single camera matching and multi camera matching (optional) are used. Single camera matching is done over the sensor’s own object database. Intuitively, if an object appears in one frame, it is very likely to appear in the next frame. Also, if a new blob is found in current frame, it is more likely to be instance of objects found in last frame than that found in frames several seconds ago. So the matching process starts from the most recent frame, and goes backward. For each frame, it finds the objects appear in that frame, then for each object that has not been matched, from most recent timestamp going backward, matches the new blob against blobs of the object in that timestamp. If at certain time, an object consists of multiple blobs, we will merge the blobs and compute the average color histogram. The

average color histogram is used when matching. In addition, “first match” is used, that means matching process stops as soon as one matched object is found. Because of the intuition given at first, first match works pretty well and a lot faster than “best match”. Figure 2 illustrates this local matching process.

Multi camera matching is used when a sensor wants to know if a new blob is instance of some object found in another sensor. It contains 3 steps. First, a filter is sent to a remote sensor. Then, the remote sensor will look for the objects using that filter, and sends them back. At last, local matching is done to find out whether there is a match between the new blob and those candidate objects. A filter can be any criteria that prune certain kind of objects, for example, objects found in last 10 seconds. A meaningful filter can be average color histogram, so that the remote sensor only looks for objects with similar average color histogram. What I am using now as the filter is the blob itself. The remote sensor looks for the candidate object using the blob (that, in fact, is local matching in remote sensor). If no candidate is found, empty object is sent back.

After matching is done, if no matched object is found, a new object will be created and marked as first observed in current sensor. The new blob is inserted into that object as instance at the time the blob is found. Then, the object with new blob will be stored (or updated if it is not a new object) in local object database. At present, we do not notify the remote sensor for object updates if matching is found remotely.

	Obj1	Obj2	Obj3
Frame1	X		
Frame2	X	X	
Frame3	X	X ③	X
Frame4	X ①		X ②
Frame5			

“X” means an instance (blob) of object is found at a frame (time).

If a new blob is found in Frame5, objects found in Frame4 will be matched first. ① Obj1 will be loaded, and its blobs will be matching from most recent frame going backward. ② Obj3 will be matching same as Obj1. If there is no match found. Objects found in Frame3 will be used. Since Obj1 and Obj3 have already been matched, so only Obj2 will be used ③.

This stops as soon as the first match is found.

Figure 2. Local matching.

For now, the sensor just floods out its remote matching requests. But in fact, a much cleverer method can be applied. With no doubt, there are some sensors located closely to each other, so that they form a group. An object found in one sensor in the group is more likely to appear in other sensors in the group than that far away. So instead of flooding out requests, one sensor can selectively choose where the requests need to be sent to. And with location, orientation and object speed information, we are able to predicate time of appearance in another sensor. Even when some time, the color histogram matching shows a less perfect match, but based on sensor location information, we are able to adjust matching rate to a little higher. We consider manually building certain kind of weighted connection graph and adopting location information in matching as a meaningful step to push forward next.

GUI

The GUI is an interface provided to let user monitor sensors and data of the network, as well as track objects on the floor map. So the displaying part of Koca’s GUI is kept with small modifications. However, logic behind that is rewritten. In fact, we want the GUI to be a supporting component of VSN, but not required. Ideally, the GUI node is a type of special sensor, which does not produce data, nor has a processing unit. It can freely join the network of sensors when necessary. In addition, there can be multiple GUIs monitoring the network at the same time but with different purposes, so GUI needs to be able to select region of interests.

The processing and storage part of the GUI is removed. Instead, an object database may be introduced for caching purpose only. Additionally, the network component is added to it so that it is able to join the network and send requests just as common sensors do. The GUI also announces itself when joining the network, so that other sensors can ask the GUI for data (this is supported by the caching object database), even it is not a data source. The GUI maintains a list of “active” sensors, and only data from those active sensors is asked for and then displayed. After the GUI has detected some sensors, GUI users are able to add them to the active list, or remove them from it. By default, locations and trajectories of all objects found in active sensors are fetched, and then a set of view-to-map coordinates will be applied to display them on the floor map. If a user chooses to see from a sensor’s view, a new window will be opened. That window will display the last frame received from

that sensor, along with bounding boxes over detected objects.

4. Three Architectural Models

Three different models have been implemented using those components. Data flows differently between the components in those models.

Push-based Model

My first demo is implemented in push-based model. The system is set up as shown in Figure 3. Each sensor has a processing unit and a local object database that stores all images captured, as well as data produced after image processing, such as color histogram, blob metadata (size, bounding box, etc.). The GUI has no processing unit but only an object database. But different from sensor's, GUI's database does not store images. When a new frame is captured, the processing unit will be used to process it. Periodically, the sensor's database pushes some of its new data to the GUI's database using the network component. Those data is mostly low-rate, which primarily consists of new objects' metadata (location, speed, etc.), but not including new frames. Frames are not pushed because of their large volume. If the GUI is not interested in them, it would be a huge waste of bandwidth and storage. The GUI periodically checks its database to update the view of floor map, but never communicates with sensors directly. If a user wants to see from a sensor's view, a separate connection will be established between the GUI and the sensor for frame transmission only, and a buffer is initialized to store frames pushed by the sensor.

This model is simple, and is good for persistent queries. However, it has some problems. For example, it is inefficient. All sensors are pushing data to the GUI, even though the GUI, in fact, is only interested in a few of them. Another concern is that the GUI is not able to ask for what it wants, or more precisely, ask for different data from different sensors, but can only select from what sensors have pushed. What we are interested in is to make the GUI able to ask questions, so the pull-based model is introduced.

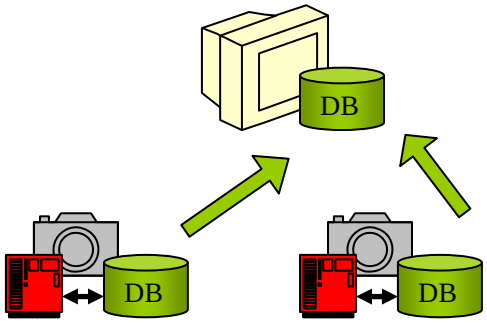


Figure 3. Pushing Model.

Pull-based Model

Pull-based model has similar setup as the push-based model. Sensors work as before except that they will not push data to GUI's database. The GUI will periodically ask for data using network component instead of being feed. So frames are not necessary to be sent over separate connections, but are transmitted same as other data. GUI's database can be used as temporary storage, so that if the database contains data up to time t1, when GUI sends request at t2, it only needs to ask for new data generated from t1 to t2 to save bandwidth.

This model is good, but if programmer needs to send those requests manually, it is a little complicated. So, later on, we considered having a data-centric implementation by providing a uniform interface so that everything is gotten from the object database. In fact, we can consider the whole sensor network as a database. Data is stored in a distributed fashion in the sensors. A sensor's local object database is not only local storage, but can also act as a view of the whole database. For example, if a sensor's processing unit is asking for objects of some other sensor. Users don't need to aware how data is fetched, but just need to ask its local database for those data as if whole network's data is stored in it. Instead, the database handles that request by fetching from its own data, checking in the cached data or forwarding the requests to other sensors. The in-memory database

component is modified to prevent users from knowing how data is shared, and I have implemented the prototype for this function in the second demo.

Pub/Sub Model

We also thought that the pub/sub model might be a good fit into our system, because it is more suitable for persistent queries and data centric implementation. In the pull-based model, one way of doing persistent query is to start a thread that repeatedly runs that query. While in pub/sub model, a consumer just need to subscribe and then wait for notification. A parameter can be specified as query last time.

We jointly implemented the pub/sub model. First, the kernel, which is called bvsn_pubsub is implemented based on Boost Signal. In fact, bvsn_pubsub can be considered as a set of data queues. It acts as connections between publishers and subscribers, and ensures data published is forwarded to all subscribers in specified order. Also, the components' behaviors are modified. When they are initializing, the components register to the kernel as publishers of some type of data. When they need some data, instead of asking corresponding components directly, they subscribe to the kernel for that type of data, so that some publisher can take over. A stub is used when subscribing, which contains procedures that a publisher needs to do for that subscriber. Figure 4 illustrates the pub/sub model.

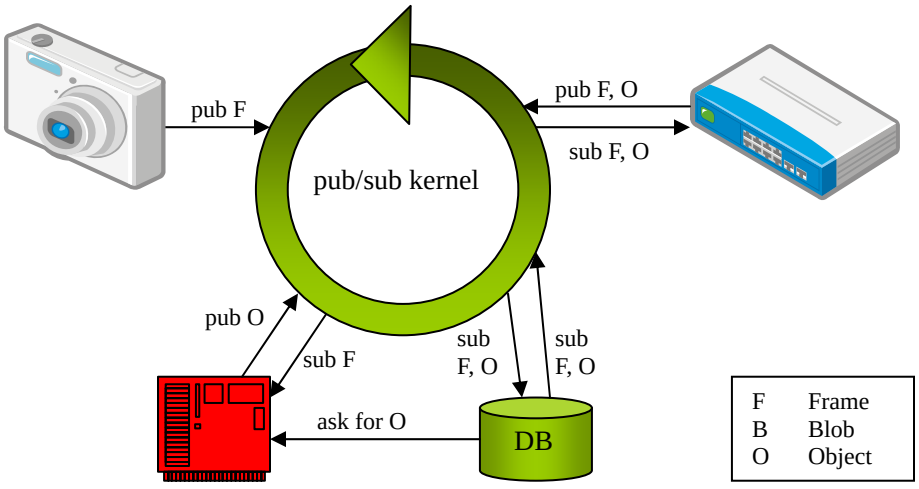


Figure 4. Pub/sub Model.

Pub/sub model can also help sensors track objects. Suppose a sensor S_A subscribes only at another sensor S_B for a new object found and wants to see the new object's movement in the whole network. Once an object is found at S_B, S_A will be notified. Then, using sensor connection graph, S_B will be able to predicate the sensors that are likely to find the object later, so that it can help S_A subscribe at those sensors by forwarding the subscribing stub. It is a more elegant way of implementing the data-centric demo because S_A does not need to figure out where data comes from, but gets what it wants.

However, programming in pub/sub model is more sophisticated, because requests and responses are not synchronized. Problems, including thread synchronization and readers-writers problem, need to be taken into consideration and handled carefully.

5. Future Work

This is only the beginning of the project. We implemented a working system that we can gradually add more functions. There are several remaining research questions to be explored. Some of them are discussed here.

Moving from 2D to 3D

We moved to network cameras because previous USB cameras setup makes us slow. But after that, we become inaccurate, not only because of low resolution, but also because network cameras forbid us from working in 3D space as they are not synchronized. One big issue arises from blob finding. In 2D space, we were

experiencing the shade problem. An object in the frame might appear as more than one blob, sometimes because of the shade on the wall or floor. Akdere has incorporated some probabilistic algorithm to figure out if one blob is actually the noise. But we are not fully in confidence. However, this would not be a problem for 3D. Also, there might be two objects that if looked at from some view angle, one is (partially) occluded by another. In this case, blob finding algorithm is only able to find one blob. While in 3D, depth information can be used to separate them. We have been looking for synchronized network camera pairs to solve those problems. But still, speed is a major concern. As for now, more than 90% of the processing time is used in imaging related processing, like finding blobs. If 3D is introduced, it will definitely be much slower if same hardware is used. On one hand, improving the algorithm is a must. On the other hand, we might need to rely on some specific hardware. Good news is NVIDIA CUDA. CUDA is the parallel compute engine in NVIDIA GPUs, that is accessible to software developers through industry standard programming languages. Yong Zhao has showed to the whole group a real-time dense matching demo which is implemented in CUDA. Adopting CUDA in our system would significantly reduce imaging processing time.

Network enhancement

Our currently TCP/IP network is mainly used to test our ideas for the whole system. But if large scale network needs to be deployed, wireless network is a must. First, wireless network has extremely critical restriction over amount of data transmitted, which introduces a need of algorithms, protocols, and programming models with a particular focus on saving network bandwidth. More work needs to be done, in order to figure out low-rate but unique features required by objects matching. Second, wireless network is not point-to-point. If a sensor needs to send some data to another sensor which is outside of its coverage, some other sensors will be involved as relay. As a result, routing protocols have to be designed, in which, power is considered as a very important factor. If not, suppose one sensor is always used as relay, its battery will be used up shortly. We need to avoid that situation. There have already been many researches in related area. Adapting them in VSNs environment would be a good option.

Supporting more queries

Currently supported queries are all built-in queries, which are provided as parts of GUI's basic functions like displaying all objects found in last frame. In the future, user specified queries need to be supported. Different from built-in queries, user specified queries are mostly in descriptive style; for example, show me objects in red. In that case, a query planner is required to translate descriptions into SQL.

Let us first focus on the GUI. If it has several active sensors and is interested in running (multiple) different queries on each of them, there are some possible ways of doing that. For the pulling model, a query list should be attached to those active sensors. So when it is time to update the floor map, in addition to built-in queries, queries of active sensors will also be executed and then, result displayed. For pub/sub model, when the GUI is subscribing itself to some sensor's data, it has to send the query. So when new data is ready, that query will be used to generate SQL and get desired result, which is indeed what the GUI subscribes for. At last, that result is published to the GUI.

In both of the aforesaid cases, the GUI knows which sensor to ask for data. But there might be more generic queries that the GUI doesn't know which to ask. And we do not want to flood out the query because of consideration of bandwidth. One possible solution is to enhance the data-centric implementation, so that the GUI just asks for its local database for result. Internally, the connection graph might be of help in predicating which sensor is the best to ask for. So that one can act as an agent and publish the query selectively in the whole network. More researches need to be done for queries like that.

Appendix A. bvsn_objectdb Schema

```
DROP INDEX IF EXISTS blob_index1;
DROP TABLE IF EXISTS BLOB;

DROP TABLE IF EXISTS BLOB_COLORHIST_DESC;

DROP INDEX IF EXISTS blob_colorhist_index1;
DROP TABLE IF EXISTS BLOB_COLORHIST;

DROP TABLE IF EXISTS BLOB_REGION;

DROP INDEX IF EXISTS frame_index1;
DROP TABLE IF EXISTS FRAME;

DROP INDEX IF EXISTS object_index1;
DROP TABLE IF EXISTS OBJECT;

DROP INDEX IF EXISTS obj_blob_index1;
DROP TABLE IF EXISTS OBJ_BLOB;

CREATE TABLE FRAME(
    cid             integer     NOT NULL,
    framenum        integer     NOT NULL,
    frame           blob,
    UNIQUE(cid, framenum)
);

CREATE INDEX frame_index1 on FRAME (cid, framenum);

CREATE TABLE BLOB(
    bid             integer     PRIMARY KEY AUTOINCREMENT,
    cid             integer     NOT NULL,
    framenum        integer     NOT NULL,
    fbid            integer     NOT NULL,
    x0              integer     NOT NULL,
    y0              integer     NOT NULL,
    width           integer     NOT NULL,
    height          integer     NOT NULL,
    npixels         integer     NOT NULL,
    UNIQUE(cid, framenum, fbid)
);

CREATE INDEX blob_index1 ON BLOB (cid, framenum, fbid);

CREATE TABLE BLOB_COLORHIST_DESC(
    bid             integer     PRIMARY KEY,
    chist_size      integer     NOT NULL,
    chist_dim       integer     NOT NULL,
    histtype        varchar(20)
);

CREATE TABLE BLOB_COLORHIST(
    bid             integer     NOT NULL,
    pos             integer     NOT NULL,
    colorhist       real        NOT NULL
);
```

```
CREATE INDEX blob_colorhist_index1 ON BLOB_COLORHIST (bid, pos);

CREATE TABLE BLOB_REGION(
        bid             integer     NOT NULL,
        ilo             integer     NOT NULL,
        ihi             integer     NOT NULL,
        j               integer     NOT NULL,
        UNIQUE (bid, ilo, ihi, j)
);

CREATE TABLE OBJECT(
        objid           integer     PRIMARY KEY AUTOINCREMENT,
        cid             integer     NOT NULL,
        local_objid     integer     NOT NULL,
        maxspeed        real        NOT NULL,
        avgsize         real        NOT NULL,
        UNIQUE(cid, local_objid)
);

CREATE INDEX object_index1 ON OBJECT (cid, local_objid);

CREATE TABLE OBJ_BLOB(
        objid           integer     NOT NULL,
        bid             integer     NOT NULL,
        UNIQUE(objid, bid)
);

CREATE INDEX obj_blob_index1 ON OBJ_BLOB (objid, bid);
```