

Effective Data Transmission in Distributed Visual Sensor Systems

Wei Song (weisong@cs.brown.edu)

Abstract:

Objective of this research project is to design an effective and stable data transmission solution for our Distributed Visual Sensor Systems. On one hand, to achieve “effective”, we need to reduce network traffic by precisely eliminating unnecessary communications among cameras; on the other hand, to achieve “stable”, we must ensure that each camera does receive its deserved data in time.

To fulfill these two points in our distributed system, after object enters/leaves certain camera, we keep track of its appearance times in other cameras with corresponding time interval, then store them to camera’s own database. Each camera takes these records as reference to select destinations of sending, and updates its database based on runtime feedback it received. Network issues including unexpected latency, redundant sending and packet missing are taken into consideration as well to ensure the accuracy of our application.

1 Introduction

In processing perspective, system is divided to two parts: offline part and online part. After setting up the cameras, system starts on offline mode: sample data of single object’s travelling is collected to a central database. Once enough data is collected, system analyzes all data centrally and generates a prime *relationTable* for each camera. System then switches to online mode: when object appears in camera A, A looks through its *relationTable*, selects proper destination cameras and then sends object’s info to them; meanwhile, feedback is sent to A’s source cameras for updating their *relationTable*. In design perspective, system is composed of three modules. Data module implements the interaction between local data structures and database; network module, which follows RIP, provides stable communication among cameras; control module takes the charge of connect all three parts together, monitor object’s moving, coordinate data processing between itself and other cameras, pick up proper cameras for sending incoming/feedback message, update local data structure and deal with potential issues.

Cameras are classified according to their viewpoints. Single camera, which has no intersection on viewpoint with others, follows a standard processing procedure. It receives potential object’s info from its source cameras, sends reply to its source cameras when object enters, spreads potential info to its destination cameras when object leaves, and updates its relation table when reply message is received. Overlap camera, which has intersection on viewpoint with others, except all procedures mentioned above, contains lots of new features to make the whole cluster work as a single camera. For example, potential object’s info should be spread inside overlap group when it enters, one and only one camera in cluster should be selected to take the charge of sending object’s info after object leave.

This paper will introduce offline part in Section 2 first, which is kind of centralizing processing and independent from application’s core modules. Section 3 reviews the data structure designed for online part. In Section 4 and Section 5,

features of single camera processing and overlap camera processing of online mode are investigated. Finally, Section 6 and Section 7 illustrates application level processing and future work.

2 Offline Part

Offline part implements centralizing processing, objects' travelling data that detected by all cameras are stored together in a central database. When enough data is collected (before switching to online mode), system analyzes these data, generates a prime *relationTable*, and then distributes proper part of copies to each camera. Since all records are stored in a central database, it's easy to get a rudimentary relation map among all cameras by sorting records by their timestamp. The only problem lies in how to make it compatible with our former database.

In former system, if two objects get to an overlap status (two or more objects have a conjunction in camera's view), a new object ID is given to represent them. For example, object 101 and object 102 enter camera A, we have records in central database's frame table for "object 101 and object 102 were in camera A" with certain timestamps. Then object 101 and object 102 walk towards each other and have a conjunction in A's view, old system takes "101 & 102" as a new object with object ID 103, we have new records for "object 103 was in camera A", information for object 101 and object 102 is temporary "lost" till these two objects separate. To get a precise map, we solve this issue by keeping an active object list (objects currently in camera) and carefully checking each record's object description of its predecessor records (early records from same camera) and its successor records (later records from same camera) when description changed.

After offline part processing, each camera receives a prime *relationTable* as following (take camera 126 as example), this table is stored in camera's own database.

Source	Destination	<0 (s)	0-20 (s)	20-50 (s)	50-90 (s)	No limit
126	91	0	0	3	2	2
126	121	3	5	3	0	0

3 Overview of Data Structure

In this section, data structures with their design purpose, definition, functionality and basic operations will be introduced. Operations across multiple data structures will be discussed in later sections with real cases.

dataView:

To reduce database's workload, we load camera's *relationTable* from database to a local data structure, **dataView**. For each camera, *dataView* stores all its destination cameras' ID, IP address and possibility (times) of appearance for both single (non-overlap, with time interval) cases and overlap cases.

It supports basic operations such as reload from database, modify *dataView* according to runtime feedback (authorized reply entry), return a list of all overlap cameras, or a list of single cameras with a specific time interval (lists are ordered by appearance/weights) and update back to database.

potentialTable & unknownTable:

potentialTable stores objects that may appear in current camera later, which is called potential objects. It records potential object's ID, object's description, object's source camera and its enter/leave timestamp in source camera. Based on *potentialTable*, camera identifies incoming objects and gets a list of source cameras to which it should send reply messages.

For example, if there is a “link” between camera A and camera B – object that leaves camera A usually appears in camera B later. When object M leaves camera A, camera A sends M’s info to camera B, camera B receives this info, validates it, then stores it in its *potentialTable* (or take other actions, explain later). When object M enters camera B, camera B finds its info in *potentialTable* along with source camera A, camera then identifies object M and sends a reply message back to A.

unknownTable stores object that is currently in camera but not identified (includes new coming object). When potential message receives, camera checks *unknownTable* for object identification, if finds, camera moves object from *unknownTable* to *activeTable*. Object that stays in unknown table exceeds a certain threshold will be taken as a new object.

potentialTable and *unknownTable* are complementary for each other. One stores objects in camera but not identified, one stores objects may appear for identification. When object enters, it checks *potentialTable*, if found, add it to *activeTable*, if nothing found, then adds to *unknownTable*; when potential message receives, it checks *unknownTable*, if found, move it to *activeTable*, if nothing found, then adds to *potentialTable*.

activeTable & broadcastTable:

activeTable stores object that is currently in camera and identified.

broadcastTable stores objects that need to be “broadcast” to other cameras. For each object, two broadcast records are created: Type 0 (E/E) for overlap cameras only, which is sent immediately when object enters (be identified), then deletes after sending. Type 1 (L/E) for all cameras, which is sent timely to high weights cameras (selected according to *relationTable*, time interval and network capacity) when object leaves, entry will be delete when related reply message is received.

There is a “link operation” between *activeTable* and *broadcastTable*. For example, when object M is added to *activeTable*, M is added to *broadcastTable* as well with type 0, which means “object M enters, broadcast it to all overlap cameras”; when object M is removed from *activeTable*, M is added to *broadcastTable* with type 1, which means “object M leaves, broadcast it to proper cameras”.

replyTable:

When incoming object finds its info in *potentialTable* and gets identified, it sends reply messages back to its source cameras (stored in *potentialEntry*) to inform them update their *relationTable*. **replyTable** is designed to store these reply messages, it ensures all these messages are correctly received by source cameras.

To achieve this point, firstly, one specific process takes the charge of timely sending these reply messages. Secondly, when source camera receives reply message, a *replyACK* message is sent back, inform destination camera to remove related entry from *replyTable*.

historyTable:

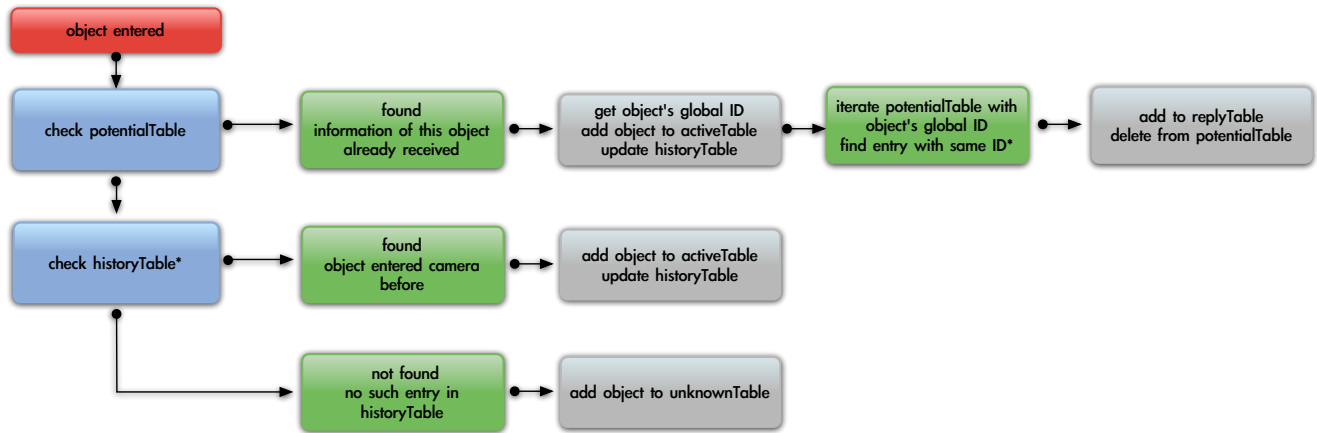
HistoryTable stores the latest status for each object that once appeared in current camera, including object’s ID, object’s description, enter/leave timestamp and an *updateTime* array. *historyTable* is designed to validate potential messages and reply messages, it handles network issue such as latency, package missing and duplicate messages.

When object gets identified, the enter/leave timestamp, together with *updateTime* array in *historyTable* are set to object’s *enterTime*; when object leaves, only leave timestamp is updated. That is to say, if enter timestamp and leave timestamp are equal, object is currently in camera. *updateTime* array is used to validate reply message, it prevents camera for accepting outdated or redundant messages.

4 Single Camera Processing

Camera that has no intersection on viewpoint with others is called single camera. Communications among single cameras are the simplest cases in our application. Mostly, packets arrive in a predicted order; cameras then search, modify and update their data structures step by step. The only matter to attention here is all about network transmission issues such as latency and package missing. In this section, Normal cases for processing of enters/leaves object will be introduced first, followed by several network issues and solutions.

Object enters camera:



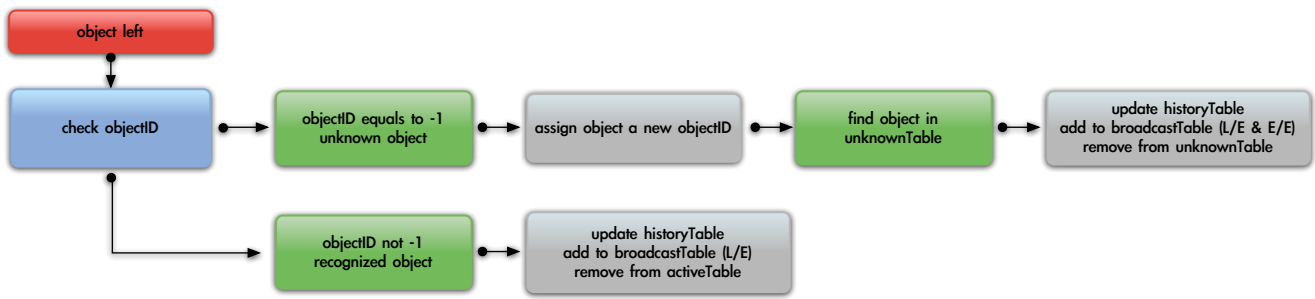
- When object M enters, camera A firstly checks its *potentialTable* with incoming object's description. If M's info is found, object M's ID will be saved, camera A starts a new iteration in *potentialTable* to find all source cameras of the same *objectID*, then creates Reply messages for each source camera, add them to *replyTable* for sending and delete related entries in potential table. Meanwhile, camera A updates its *historyTable* of entry M, add M's info to *activeTable*, then return.
- Secondly, if nothing is returned from *potentialTable*, camera A checks its *historyTable*. If found, camera A updates its *historyTable*, camera adds M's info to *activeTable* then return; if not, camera A adds M to *unknownTable* for further processing then return.

Comments:

1, Since it's more convenient to compare between integer values rather than huge strings, system uses object's ID instead of description for a new iteration in *potentialTable* to retrieve all objects with same object ID.

2, Generally speaking, *historyTable* shouldn't be used for object checking, but one case may happen. If object M leaves A then re-enters A again, there is no info about M in A's *potentialTable* (M's records are deleted when it enters first time), but A shouldn't be stored in *unknownTable* and be identified as different object later. Under this circumstance, *historyTable* is used for complementary object identification.

Object leaves camera:



- When object M leaves, camera A checks its object ID first. If M's object ID equals -1, M is an unidentified object and currently stored in *unknownTable*. Camera A takes M as a new object, assigns M a new object ID, deletes M from *unknownTable*, updates *historyTable*, adds M's info to *broadcastTable* with both type 0 and 1 step by step, then return.
- If M's object ID isn't equal to -1, M is identified and currently stored in *activeTable*. Camera A deletes it from *activeTable*, updates *historyTable*, adds M's info to *broadcastTable* with type 1 then return.

Aside from those normal cases mentioned above, since all transmissions are under an unreliable network, each camera should take network issues into consideration. Several cases are discussed in the following.

Outdated potential message:

Due to unexpected network latency, before “all” potential messages are received, object may already arrive, get processed and even leave from current camera. For example, consider the situation that object M enters camera A, camera B and camera C in sequence.

- Camera C receives potential message (M) from A, stores it in C's *potentialTable*.
- Object M enters C and finds its info in C's *potentialTable*, object M gets identified, camera deletes M from its *potentialTable*.
- Object M leaves camera C.
- B's potential message (M) reaches camera C.

Although there is no M's info in both C's *potentialTable* and *unknownTable*, potential message (M) from B shouldn't be stored in C's *potentialTable* (M already showed up in C) or just abandoned (B should know M's appearance in C). In contrast, reply message of M must be generated and sent back to B for updating B's *relationTable*. Here we use *historyTable* to achieve this point. As mentioned before, *historyTable* stores latest enter/leave timestamp of object that once appeared in current camera. When potential message is received, camera checks *historyTable* to find out whether this object already appears after it leaves source camera, if object already appeared, camera directly sends a reply message back to source camera; if not, camera checks *unknownTable* or stores it in *potentialTable*. In above example, after object M entered and left, both enter timestamp and leave timestamp of M are updated (or added) in C's *historyTable*. When B's outdated potential message (M) reaches, C checks its *historyTable* and finds a larger enter timestamp compared to B's leave timestamp, system realizes it's a outdated but validated potential message, reply message is generated and sent back to B.

Duplicate/invalid reply message:

Due to unreliable network, both reply message and *replyACK* message may lose during transmission. To solve this issue, on one hand, *replyACK* message is required to confirm the receiving of reply message; on the other hand, some work is

essential to deal with lost of *replyACK* message, which may lead to duplicate reply messages and redundant update on source camera's relation table. Consider the situation object M appears in camera A and camera B in sequence.

- Camera B receives potential message (M) from A, stores it in B's *potentialTable*.
- Object M enters B, finds its info in *potentialTable* and sends reply message back to A.
- Camera A receives reply message, updates its relation Table and send *replyACK* back to B.
- *replyACK* message lost, B resents reply Message later.
- A receives reply message from B again.

A receives the same reply message (M) twice from B. For the second time, A should refuse B's reply message and avoid redundant update to its *relationTable*. To achieve this, A maintains an *updateTime* array that stores M's latest n (current 3) timestamps of entering other cameras after it leaves A. For each reply message receives, A checks its *destETime* with *updateTime* array. If a equal timestamp is found, reply message is taken as a duplicate one, A refuses this message and sends *replyACK* message back to B; If not found, A modifies its *updateTime* array to reply message's *destETime*, accepts this reply message and send *replyACK* message back to B.

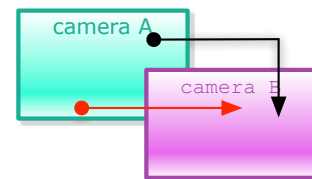
Comments:

updateTime array is renewed when object re-enters camera (set to enter time).

5 Overlap Camera Processing

A group of cameras that have intersections on their viewpoints is called a cluster of overlapped cameras. As mentioned above, to other cameras out of this group, a cluster behaves as a single camera. When object M enters cluster and gets identified, all cameras in cluster are informed ASAP; when object leaves, one and only one camera is selected to take the charge of "broadcast" M's info to outside cameras. In this section, communication issues among overlap cameras will be discussed first, followed by analysis with real cases and solutions to solve these issues in a distributed system.

Firstly, communications inside cluster inherit all features of single camera's case. Partially intersection on viewpoints means no intersection on other parts, for two overlapped cameras A and B, object M may show up in both of them at the same time (red line), or leave A then enter B later (black line). To ensure the correctness of inner communication, all rules of single camera's processing must be followed.



When object M leaves A, M's info should be sent to B according to A's *relationTable*; when object enters B, *replyACK* message is sent back to A to stop M's spreading in A and update A's relation table.

Secondly, to keep object's constancy, extra messages with higher sending priority are required inside cluster. Especially for new object (system treats object as new if it stays in *unknownTable* for a certain time, or leaves camera without a valid object ID), these messages are essential to avoid other cameras in cluster identify it as a different object. To achieve this point, once object gets identified, its info must be spread inside cluster ASAP in forms of potential message. Here comes the first issue, processing of potential message.

Processing of potential message.

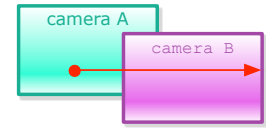
In cluster, potential message is sent twice, when object gets identified, potential message is spread among the entire cluster, which is named *broadcastEE* (enter & enter) entry; when object leaves camera, potential message is sent to selected destination cameras, which is named *broadcastLE* (leave & enter) entry. For camera, along with the first point, it may receive potential messages from same camera twice of different type. Also, without the help of direction parameter,

potential message may be sent back to cameras, which it just leaves from. That is to say, camera should be able to make out which potential message is acceptable, which potential message should be refused. Classification & validation part for potential messages is designed to solve these issues.

1, Ignore incoming potential message from source camera.

- Object M leaves camera A first.
- Object M leaves camera B, camera B sends potential message (L/E) to camera A.

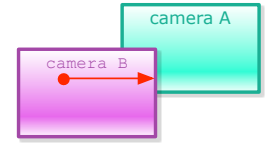
Mostly, M is walking in opposite direction towards A and won't enter A again; even if M turns around, potential message (E/E) will be sent to A when M re-enters B first. In this case, A should ignore B's potential message (L/E).



2, Ignore incoming potential message from source camera, send reply back to stop source camera's broadcasting.

- Object M enters camera B, then enters camera A.
- Object leaves camera B, B sends potential message (L/E) to A, which M is still in.

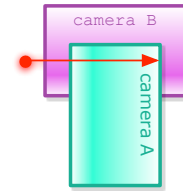
In this case, object still in A, which means camera A now takes the charge of broadcasting M's info. A should ignore this message and send a feedback to B for stopping M's broadcasting in B.



3, Accept incoming potential message from source camera and find related info in destination camera's *unknownTable*, update destination camera's own relation table or send reply to update source camera's relation table.

- A new object enters B, B stores object's info in its *unknownTable*.
- New object enters A and leaves camera A, object is taken as a new object M.
- A sends potential message (L/E) back to B.

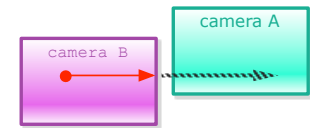
In this case, object M gets identified in camera B when it receives potential message from A. Since M enters B earlier and is still in B, B should update its own relation table, then sends a reply message back to A to stop M's broadcasting in A.



4, Accept incoming potential message from source camera, if object enters in future, send reply back to stop source camera's broadcasting and update its relation table.

- Object M leaves camera B, B sends potential message to A.
- Object M is still on its way to camera A.

In this case, A should accept B's potential message. If M enters A later, A will send reply message back to B to inform M's coming, stop M's broadcasting in B and update B's relation table.



In implementation, take network latency & packet missing (when object appears in camera, parts or all potential messages are still on their way) into account, to cover all possibilities, totally 16 cases are counted. 10 cases are for checking *historyTable*, 5 cases are for checking *unknownTable* and 1 left is for adding to *potentialTable*. Codes for each case can be found in function *processingPotentialEntry*, Graphs and descriptions can be found in Appendix A.

Thirdly, to ensure cluster acts as a single camera, when object M leaves cluster, one and only one camera in cluster is selected to take the responsibility of sending M's info to others outside cluster. In distributed system, when object M leaves cluster, the camera, which M leaves latest, is chosen as such an "administrator" camera. While object is still in camera, we also need an administrator camera to control others' sending, in this phase, "administrator" camera is selected dynamically, camera that currently holds object M works as an administrator, when it receives potential

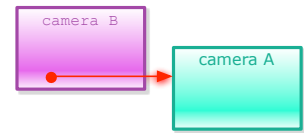
message (*broadcastLE* entry) from other cameras in cluster, it sends reply back to announce “stop sending M, I take charge of M now”, in forms of reply message. That is to say, aside from the functionality of informing update, in some cases reply message is used to inform source camera to delete broadcast entry only. The second issue emerges, how to differentiate varieties of reply messages.

Pre-processing of reply message.

As we discussed in processing of single camera, reply message is used to inform camera to update its *relationTable* & delete related potential entry in *broadcastTable*. Now, under some circumstance, reply message is used to stop source camera's broadcasting only. Pre-processing part for reply messages is designed to solve these issues.

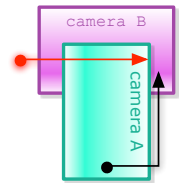
1, Delete broadcast entry & update relation table.

- Object M leaves B, B sends potential message (L/E) to A. A receives B's potential message and store it in A's *potentialTable*.
- Object M enters A, A finds related info in *potentialTable*, then sends reply message back to B. B receives A's reply message, delete M from its *broadcastTable* and update its *relationTable*.



2, Delete broadcast entry only.

- Object M enters B, B sends potential message (E/E) to A.
- Object M enters A, A sends reply message back to B. B receives A's reply message accepts it.
- Object M leaves A, since M may enter B after leave A (black line), A sends potential message (L/E) to B. B receives A's potential message, send reply message to inform A stop broadcasting.
- A receives B' reply message; delete M's info in its *broadcastTable* only (no duplicate update).



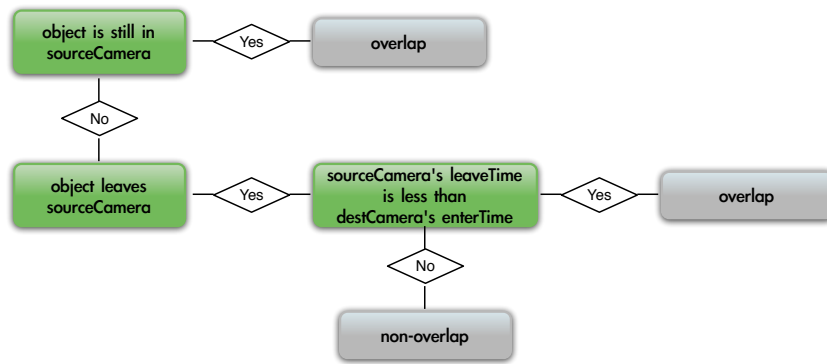
Not as the same as classification & validation of potential message, which includes 16 cases, there are only 3 cases for pre-processing of reply message: delete & update, delete, ignore (duplicate reply, discussed in processing of single camera section). The truth lies in the sequence of processing: classification & validation part of potential message eliminates several cases for pre-processing of reply message in advance, it also makes processing of reply message easier, which will be discussed later.

6 Application level Processing

After the independent introduction of single camera and overlap camera, along with the discussion of several issues so far, a rudiment map of this application is out. Now we combine them together and move discussion to application level.

Processing of reply message

After the pre-processing of reply message, which filters lots of illegal cases, camera takes both *historyTable* and reply message as reference to update its *relationTable*. If overlap, update its overlap column, if not, update its non-overlap column, based on exact time interval.



- If object is still in source camera, which means object still in source camera when enters destination camera, overlap.
- If object leaves source camera, check the relation between time that object leaves source camera and time that object enters destination camera. If object enters destination camera earlier, update overlap column. If objects leaves source camera earlier, update non-overlap column with time interval $(destETime - sourceLTime)$.

Priority hierarchy of sending

Till now, camera has four different kinds of message for sending. Details of each kind with its priority are in the following:

Potential message (E/E): high priority. When object enters (be identified), camera spreads potential message (E/E) inside cluster immediately to prevent other cameras in cluster taking the same object as a different one. Potential message (E/E) is stored in *broadcastTable* with type 0, deleted after sending.

ReplyACK message: high priority. Camera sends reply message to stop its destination camera sending reply message that already received by source camera, network traffic will be reduced effectively. *ReplyACK* message will be send immediately when reply message is received and validated.

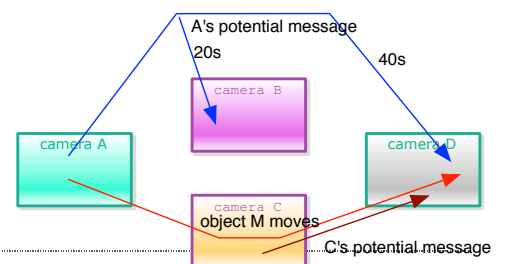
Reply message: medium priority. On one hand, reply message is used to update *dataView*, on which all our broadcasting operations based, to provide a more precise destination list. On the other hand, reply message prohibits source camera to send potential message (L/E). Reply message is stored in *replyTable*, deleted when related *replyACK* message is received.

Potential message (L/E): low priority. Camera sends potential message (L/E) to selected destination cameras when object leaves camera. Since it usually takes seconds, even minutes between object leaves current camera and enters destination camera, potential message (L/E) has the lowest priority. Potential message (L/E) is stored in *broadcastTable* with type 1, deleted when related reply message is received.

Recall amendment:

Under runtime observation, following case may happen.

- Object M leaves camera A, due to network capacity at that time, A is allowed to send only one potential message. A sends potential message (M) to camera B, which has a higher possibility than C when time interval is 20s.



- Object M enters camera C (stored in *unknownTable*). When M leaves C, it is taken as a new object name N. C sends potential message (N) to camera D.
- Since A doesn't receive any reply message, it sends potential message to camera D, which has the highest possibility than others when time interval is 40s.
- Object M enters D. Now in D's *potentialTable*, M (from A) and N (from C) represent the same object.

In earlier implementation, since camera uses object ID to retrieve source cameras, to which destination camera sends reply messages. In this example, either M or N gets processed depends on their position in *potentialTable*. Here, camera D not only needs to reply to both of them, but also sends extra reply message to A on behalf of C. To achieve this purpose, *replyVector* is used when processing enter object.

When object enters camera D, all entries in *potentialTable* is checked by comparing its object description with incoming object's description. If validated, camera D inserts that potential entry into *replyVector*, sorted by enter time (*replyVector* contains M (from A), N (from C)). Then camera D iterates *replyVector* from its beginning, for each entry in *replyVector*:

- Step 1: Camera D sends reply message to current entry's source camera (D sends reply message to A).
- Step 2: Camera D checks its following entries, if object ID in following entries is different from current entry, D sends reply message to current entry's source camera again on behalf of that following entry. (D sends reply message to A on behalf of C).
- Step 3: Camera D deletes current entry. If *replyVector* is empty, return; if not, go to step 1.

Simply speaking, according to its *potentialTable*, camera D reconstructs object M's path before it enters D. Then camera D gets to know to which it should send an extra reply message. Along with *updateTime* array in *historyTable*, no redundant update will happen.

Fault tolerance:

Like other distributed systems, when one or some nodes fail, system should work it out and keep working.

On network level, application implements Routing Information Protocol (as Jie suggested), nodes send routing-update message at regular intervals and when the network topology changes. When one node fails, or path weight between two nodes changes, each node's routing table updates on cascade, new shortest path is found and the whole network returns to normal.

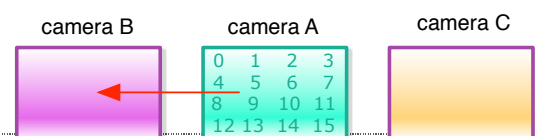
On control level, if destination node fails, for reply message and potential message (E/E & L/E), both of them are stored in table, sent timely and deleted when related feedback comes. An expiration time threshold is set for both tables, after a certain time, entries without receiving feedback expire and get deleted. For *replyACK* message, it is sent when destination camera's reply message received, since destination node already failed, no more reply message comes.

On database level, *dataView* is updated back to database timely, by which information loss of node failure reduces to minimum.

7 Future Work

Partition camera scene

The idea of partition scene comes from Jie. In his suggestion, scene of camera is divided into 16 rectangles (4 x 4), marked them from 0 to 15. By doing this we figure out a method to indicate object's position in



camera, through which we can improve our application. For example, by tracing the position object leaves, camera can select its destination camera more precisely: object leaves camera A with grid numbers (4, 8), then camera knows object leaves from its left side, it sends potential message (L/E) to camera B instead of C.

This idea comes to me when offline part was totally done, online part is in progress and mostly completed, so I marked it as further work. To achieve this, firstly, almost all data structures should be changed. Secondly, I need to figure out a way for A to select its destination cameras based on both grid info and *relationTable* while taking network issue together into consideration.

Network Capacity:

As discussed in “Priority hierarchy of sending, Section 5”, application has four kinds of messages for sending. The first three, *potential message (E/E)*, *replyACK message* and *reply message* are essential for either ensuring correctness of system, or informing camera to stop sending something and reduce network traffic, or both. These three kinds of messages can’t be skipped even network is busy.

The only kind of message that can be altered according to network condition is *potential message (L/E)*. Camera can decide the number of its destination cameras for sending. For example, if network is busy, camera sends *potential message (L/E)* to top 2 destination cameras; if network is free, camera sends it to top 5 destination cameras.

Number of destination cameras should be dynamically decided base on some parameters that can precisely reflect the current network capacity, I used to take “number of redundant reply message it received” for this functionality, just like what we do in TCP, each time a redundant ACK message is received, cut the window by half and increase by 1. But reply message mostly works independent with potential message, and then it can’t cover all cases. Now this number is hard-coded, I’m seeking for a better one to reflect runtime network capacity.






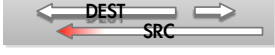
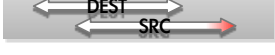






8 Conclusion

In summary, while ensuring system’s correctness, application discussed above effectively eliminates unnecessary packet sending and reduces network traffic. It also provides a proper solution for coordinating the data processing in overlap cameras, which keeps object’s consistence and makes the distributed overlap camera cluster work as a whole. Besides these, several solutions are discussed & implemented to deal with potential errors caused by unexpected network issues.

9 Acknowledgement

I’d like to take this opportunity to thank my advisor Professor Ugur Cetintemel for his support in the past two years, especially for his concern during my surgery. I would also like to thank Jie Mao and Mert Akdere for their patient help and valuable suggestion for my work.

Appendix A: all cases of classification & validation part for potential messages

check historyTable		
Case 1: SRC camera sends potential message when object leaves.		DST camera refuses this potential message.
Case 2: SRC camera sends potential message when object leaves.		DST camera sends reply to stop SRC's broadcasting.
Case 3: SRC camera sends potential message when object leaves.		DST camera sends reply to stop SRC's broadcasting.
Case 4: SRC camera sends potential message when object enters.		DST camera sends reply to stop SRC's broadcasting and update SRC's table.
Case 5 & 7: SRC camera sends potential message when object leaves.		DST camera sends reply to stop SRC's broadcasting.
Case 6 & 8: SRC camera sends potential message when object enters.		DST camera refuses this potential message and update DST's table
Case 9: SRC camera sends potential message when object leaves.		DST camera refuses this potential message.
Case 10: SRC camera sends potential message when object leaves.		DST camera refuses this potential message and go to next step.
check unknownTable		
Case 1: SRC camera sends potential message when object leaves.		DST camera sends reply to stop SRC's broadcasting and update SRC's table.
Case 2: SRC camera sends potential message when object enters.		DST camera sends reply to update SRC's table.
Case 3: SRC camera sends potential message when object leaves.		DST camera sends reply to stop SRC's broadcasting and update SRC'S table.
Case 4: SRC camera sends potential message when object leaves.		DST camera sends reply to stop SRC's broadcasting and update DST's table.
Case 5: SRC camera sends potential message when object leaves.		DST camera sends reply to stop SRC's broadcasting and update DST's table.
add to potentialTable		