Federated DyKnow, a Distributed Information Fusion System for Collaborative UAVs

Fredrik Heintz and Patrick Doherty
Dept. of Computer and Information Science
Linköping University, 581 83 Linköping, Sweden
{frehe, patdo}@ida.liu.se

***Abstract*—As unmanned aerial vehicle (UAV) applications are becoming more complex and covering larger physical areas there is an increasing need for multiple UAVs to cooperatively solve problems. To produce more complete and accurate information about the environment we present the DyKnow Federation framework for distributed fusion among collaborative UAVs. A federation is created and maintained using a multi-agent delegation framework which allows high-level specification and reasoning about resource bounded cooperative problem solving. When the federation is set up, local information is transparently shared between the agents according to specification. The work is presented in the context of a multi-UAV traffic monitoring scenario.** [1]

***Index Terms*—Distributed information fusion, multi-agent systems, autonomous systems.**

In many robotic applications, having one agent execute a mission is not sufficient. In a UAV application for example, there is sometimes no single UAV that has the capability or the information to perform all required tasks. In many cases, it is also more efficient to use multiple UAVs to complete a mission. Therefore it would be beneficial for groups of UAVs to accomplish complex missions in a cooperative manner. One class of such applications is multi UAV surveillance missions, where a team of UAVs creates and maintains situation awareness by keeping track of important objects and events.

Our goal is to create a framework for distributed information fusion where groups of UAVs collectively collect and process information to achieve situation awareness.

Conventional approaches to fusing information have focused on collecting information from distributed sources and processing them at a central location. Our aim is to allow each platform to be autonomous and do as much processing as possible locally even when there is a need to cooperate to solve a particular task. This will make the processing more decentralized and remove the dependence on a central node with global information.

As part of our ongoing research in unmanned aircraft system technologies [1], [2] we have developed DyKnow, a stream-based knowledge processing middleware framework which provides design and software support for developing applications integrating sensing and reasoning [3], [4]. We have previously [5] shown how DyKnow can be used to implement the JDL Data Fusion Model, which is the de facto standard functional fusion model [6], [7]. This shows that DyKnow provides an appropriate basis for sharing and fusing high level information. In this paper we extend DyKnow to support federated information processing among collaborative UAVs.

[1]This work is partially supported by grants from the Swedish Foundation for Strategic Research (SSF) Strategic Research Center MOVIII, the Swedish Research Council (VR) Linnaeus Center CADICS, ELLIIT Excellence Center at Linköping-Lund for Information Technology, the Swedish Research Council (VR) project 2009-3857, and the Center for Industrial Information Technology CENIIT.

I. A Multi UAV Traffic Monitoring Scenario

Assume that two or more UAVs are given the task of monitoring an urban area for traffic violations. Each UAV is equipped with the appropriate sensors and reasoning mechanisms for detecting traffic violations. This means that each UAV could monitor and detect traffic violations by itself, if it sees the whole situation. We have previously shown how this could be done [8].

To increase the size of the monitored area, to improve the quality by reducing the uncertainty, or to monitor several different potential traffic violations at the same time, several UAVs can be used. However, cooperation, like dividing the area between the UAVs, introduces issues related to sharing and fusing information.

One issue is the possibility of a traffic violation beginning in one sub-area and ending in another. In this situation, neither of the UAVs will see the whole event. To handle this situation the UAVs need to cooperate and share information in such a way that they can detect the traffic violation together. One approach is to let the UAV that detected the beginning of the potential violation request the appropriate information from the UAV responsible for the area where the vehicles are headed. What is appropriate will depend on how traffic violations are detected. One approach could be to share the position information about the tracked vehicles. This information would have to be seen as a stream since it is not a single piece of information but rather an evolving description of the development of a complex situation. Fusing such a stream with local information would allow the first UAV to detect the traffic violation even if it takes place in two different areas.

This traffic monitoring scenario is an instance of a class of scenarios where multiple platforms must cooperate to complete complex missions. To succeed they need to collect, share, and fuse information. A solution which handles the issues introduced in this scenario will also provide a solution for many other interesting scenarios. For example, instead of having homogeneous platforms covering different parts of an area there could be heterogeneous platforms with complementing

978-1-4244-7815-6/10/$26.00

sensors each providing different types of information. Another example is to increase the accuracy in the monitoring by having several homogeneous or heterogeneous platforms covering the same area. It is also possible to replace traffic monitoring with searching for injured people in a rescue mission or looking for troops and equipment in a military surveillance mission.

II. DyKnow

DyKnow is a fully implemented stream-based knowledge processing middleware framework providing conceptual and practical support for structuring knowledge processing systems as sets of streams and computations on streams [3], [4]. Input can be provided by a wide range of sources on many levels of abstraction, while output consists of streams representing for example objects, attributes, relations, and events.

Knowledge processing for a physical agent is fundamentally incremental in nature. Each part and functionality in the system, from sensing to deliberation, needs to receive relevant information about the environment with minimal delay and send processed information to interested parties as quickly as possible. Rather than using polling, explicit requests, or similar techniques, we have therefore chosen to model and implement the required flow of data, information, and knowledge in terms of *streams*, while computations are modeled as active and sustained *knowledge processes* ranging in complexity from simple adaptation of raw sensor data to complex deliberative processes.

Streams lend themselves easily to a *publish/subscribe* architecture. Information generated by a knowledge process is published using one or more *stream generators*, each of which has a (possibly structured) *label* serving as an identifier within a knowledge processing application. Knowledge processes interested in a particular stream of information can subscribe to it using the label of the associated stream generator, which creates a new stream without the need for explicit knowledge of which process hosts the generator. Information produced by a process is immediately provided to the stream generator, which asynchronously delivers it to all subscribers, leaving the knowledge process free to continue its work. Using an asynchronous publish/subscribe pattern of communication decouples knowledge processes in time, space, and synchronization, [9], providing a solid foundation for distributed knowledge processing applications.

Each stream is associated with a declarative *policy*, a set of requirements on its contents. Such requirements may include the fact that elements must arrive ordered by valid time, that each value must constitute a significant change relative to the previous value, that updates should be sent with a specific sample frequency, or that there is a maximum permitted delay. Policies can also give advice on how to satisfy these requirements, for example by indicating how to handle missing or excessively delayed values.

A knowledge processing application in DyKnow consists of a set of knowledge processes connected by streams satisfying policies. Each knowledge process is either an instantiation of a *source* or a *computational unit*. In the first case, it makes

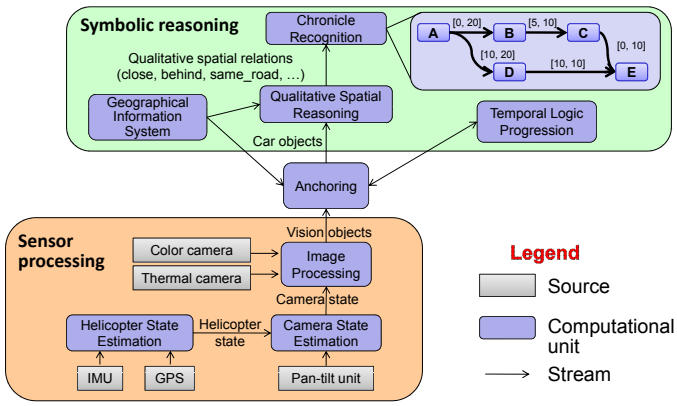

Figure 1. An overview of how the incremental processing required for a traffic surveillance task could be organized.

external information available through a stream generator, and in the second it refines and processes streams. A formal language called KPL is used to write declarative specifications of DyKnow applications (see [4], [10] for details). KPL provides a formal semantics for policies and streams. The DyKnow service, which implements the DyKnow framework, sets up the required processing and communication infrastructure for a given set of KPL declarations. Through the use of CORBA, knowledge processes are location-unaware supporting distributed applications running on multiple computers.

Fig. 1 provides an overview of how part of the incremental processing required for a traffic surveillance task can be organized as a set of distinct DyKnow knowledge processes.

At the lowest level, a *helicopter state estimation component* uses data from an *inertial measurement unit* (IMU) and a *global positioning system* (GPS) to determine the current position and attitude of the UAV. A *camera state estimation component* uses this information, together with the current state of the *pan-tilt unit* on which the cameras are mounted, to generate information about the current camera state. The *image processing component* uses the camera state to determine where the camera is currently pointing. Video streams from the *color* and *thermal cameras* can then be analyzed in order to generate *vision objects* representing hypotheses about moving and stationary physical entities, including their approximate positions and velocities.

To describe a complex event such as a traffic violation, a *chronicle* is used [11]. A chronicle is a description of a generic scenario whose instances we would like to recognize. It is defined by a set of events and a set of metric temporal constraints between the occurrence time of these events. Symbolic formalisms such as chronicle recognition require a consistent assignment of symbols, or identities, to the physical objects being reasoned about and the sensor data received about those objects. Image analysis may provide a partial solution, with vision objects having symbolic identities that persist over short intervals of time. However, objects temporarily being out of view or changing visual conditions lead to problems that image analysis cannot (and should not) handle. This is the task of the anchoring system, which uses *progression*

of formulas in a metric temporal logic to evaluate potential hypotheses about the observed objects. The anchoring system also assists in object classification and in the extraction of higher level attributes of an object. For example, a *geographic information system* can be used to determine whether an object is currently on a road or in a crossing. Such attributes can in turn be used to derive relations *between* objects, including *qualitative spatial relations* such as $\mathsf{beside}(car_1, car_2)$ and $\mathsf{close}(car_1, car_2)$. Concrete events corresponding to changes in such attributes and predicates finally provide sufficient information for the *chronicle recognition system* to determine when higher-level events such as traffic violations occur.

III. Requirements

When designing a framework for distributed information processing several important issues must be considered.

First, how to refer to a piece of information when communicating with other nodes, i.e. how to handle naming issues. This is the problem of how to agree on a common ontology among a group of nodes. The ontology is required for a node to be able to refer to a particular piece of information when talking to other nodes.

Second, how to discover information among a group of nodes. When a node needs a specific piece of information that it does not have, then it needs to find another node which is able to deliver it. Such a mechanism should be able both to find a node who either has or can produce a particular piece of information and to announce to interested nodes when a particular piece of information is available.

Third, how to negotiate with other nodes to make them generate desired information. In its simplest form this mechanism would request the production of a piece of information from a node. In the general case a node could refuse to perform the request due to limited resources or conflicting commitments. There might also be several nodes that could produce the same information but with different quality and costs. In this case, a node would have to reason about the different options and negotiate with the nodes to find one to produce the information with good enough quality while reducing the cost.

Fourth, how to deliver information from one node to one or more other nodes interested in the information. The mechanism should allow for robust and efficient transfer of information between nodes while taking the properties of the communication medium into account, such as the risk of losing or corrupting messages, low bandwidth, or a single shared channel.

To make these requirements more explicit three use cases are presented. Together they cover most of the functionality required for the multi UAV traffic monitoring scenario. The DyKnow Federation framework supports each of these.

1) Explicit Ask and Tell to Divide the Monitoring Area: To divide an area to be monitored among a group of UAVs they need to negotiate. One approach would be to appoint one of them the leader. This leader then has to find out which UAVs are available and collect information about them. The information could for example be available sensors and the maximum speed and flying altitude. Using this information the leader can partition the area among the UAVs and inform them about their responsibilities.

This use case gives an example where a node needs to find which other nodes are available, ask for specific pieces of information from each of the nodes, compute the result, and then inform the other UAVs about the result.

2) Continuous Information Streaming to Detect Traffic Violations: When monitoring a traffic violation occurring in two adjacent areas covered by different UAVs there will be an interval where none of the UAVs has a complete picture of the situation. Therefore they have to cooperate to observe the whole development. For example, when UAV A has detected the beginning of a potential traffic violation involving two cars and one of the cars moves from the view field of UAV A to the view field of UAV B. Then, UAV A has to continuously get updates from UAV B about the car to complete the detection.

The information provided by UAV B could be on many levels of abstraction. A high abstraction level in this case could be to send a stream of car states with the best current estimation of the position of the car. UAV A can then fuse the information received from UAV B with the information gathered by its own sensors in order to monitor the potential traffic violation. It is important to notice that this is an ongoing activity where each new car state computed by UAV B should be transmitted to UAV A to be merged with information produced locally. Another example is to share and fuse information on a lower level. The lowest possible level would be to send raw sensor data, such as images. This will in most cases not be appropriate since communication bandwidth is limited.

To find an appropriate abstraction level for the communication many factors must be taken into account. The most important ones are the processing capability of the involved platforms, the available bandwidth, and the current commitments of the involved platforms. Note that in this example, both UAVs were assumed to have identical abilities. In the general case, heterogeneous processing capabilities may affect the appropriate abstraction level for sharing data.

3) Fusing Information to Get a Global Picture: A slightly different use case is if an operator would like to have information about all tracked vehicles in an area. In this case, a number of UAVs are looking for and tracking vehicles. Each UAV creates its own local identifiers for the vehicles it has found. When a UAV detects a vehicle it has not seen before, it should be reported to the operator. As long as the UAV is tracking the car the operator should receive continuous updates about the estimated car state.

One question is now whether the vehicle found by the UAV is the same as one of the vehicles the operator already has information about. To fuse the information from all the available nodes it is therefore necessary to reason about the identities of the tracked vehicles. Which are the same? If two identifiers refer to the same vehicle then the information related to these identifiers should be fused.

This use case can be extended by adding and removing nodes. Each time a new node is added, then any vehicle which is tracked by that node should be reported back to the operator.

If a node is removed then the operator should be notified that the information from that UAV is no longer available.

IV. Sharing Information using DyKnow

From the point of view of DyKnow, multiple physical platforms could be viewed as sharing a single instance of DyKnow, since it is designed for a distributed environment and does not differentiate between streams based on where they are hosted or generated. However, much of the information processed by DyKnow will be local to a single platform. It would therefore incur an overhead to communicate with a single central DyKnow instance. With multiple DyKnow instances, one for each platform, this overhead is avoided. This also reduces the coupling between the nodes which makes it easier to add new nodes and to implement nodes independently.

Another benefit with having many DyKnow instances is that only relevant information needs to be shared among the instances. This is appropriate since the internal structures and representations used by one node should not necessarily be public to all other nodes. Most of it will be irrelevant and some should even be kept secret. By only sharing relevant information, communication overhead is further reduced and the robustness is increased since the system does not require reliable and stable communication all the time.

We have therefore extended DyKnow to allow different DyKnow instances to be developed and used independently, and then connected in a federation on demand. When nodes are connected, parts of their local DyKnow instances are shared.

A. DyKnow Federation Overview

To fulfill the requirements introduced in Section III we propose to connect nodes having local DyKnow instances in a DyKnow Federation, similar to the concept of federated databases [12], [13]. The federation is used to find other DyKnow instances which can provide particular pieces of information and to ask queries about information available at other nodes. To support efficient continuous streaming of information between nodes we propose to create direct communication channels on-demand between pairs of nodes. These channels are set up through the federation framework but are then under the control of the participating nodes. From the perspective of a local DyKnow instance information from remote nodes is treated as if it were local.

The DyKnow Federation framework uses an existing multi-agent framework [14], where each DyKnow instance becomes a service on its platform. A DyKnow Federation is managed through speech act-based interactions between these services.

B. The Multi-Agent Framework

An agent is a reactive, proactive, and social entity with its own thread of control. Agents communicate with each other using the standardized agent communication language FIPA ACL [15], based on speech acts. An agent provides a set of *services*. A service encapsulates a set of tasks that an agent can do. A physical platform, such as a UAV, often hosts many different agents. Each agent is FIPA compliant and is implemented using the Java Agent Development framework JADE [16].

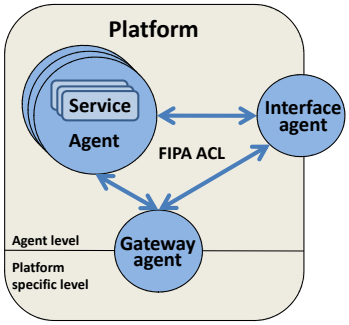

Figure 2. An overview of a platform in the delegation framework.

To support cooperative goal achievement among a group of agents a delegation framework has been developed [14]. It provides a formal approach to describing and reasoning about delegating goals and plans to other agents. The concept of delegation allows for studying not only cooperation but also mixed-initiative problem solving and adjustable autonomy.

Each UAV platform has an agent layer consisting of a set of agents communicating using FIPA ACL and a layer with platform specific functionalities (Figure 2). The interface between the two layers is the *Gateway Agent*, which provides a FIPA ACL interface to the platform specific level. In our UAV platform, where the platform specific software is implemented using CORBA, this involves invoking methods on different CORBA objects.

All communication between a platform and agents external to the platform goes through the *Interface Agent*. The Interface Agent provides a single entry point which makes it possible to keep track of communication, authenticate incoming messages, and perform access control to the platform.

A service can either be public, protected, or private. A public service can be used by any agent on any platform. A protected service can be used directly within a platform but only indirectly through the Interface Agent by an agent on another platform. Private services can only be used by agents on the same platform.

To find services in the agent framework a *Directory Facilitator* (DF) is used. It is a database containing information about available services and their providers. There is a local Directory Facilitator on each platform which keeps track of the protected and private services on the platform and a global Directory Facilitator for keeping track of all public services in the multi-agent system.

C. DyKnow Federation Components

A platform taking part in a DyKnow Federation should have three components: A DyKnow Federation service, an Export Proxy, and an Import Proxy. A DyKnow Federation service is a protected service which allows a local DyKnow instance to take part in a DyKnow Federation. The Export and Import Proxies are used to mediate streams through direct communication between two DyKnow instances. Apart from these DyKnow Federation specific components, the framework also uses the Interface and Gateway Agents. The Interface Agent is used to communicate with other platforms

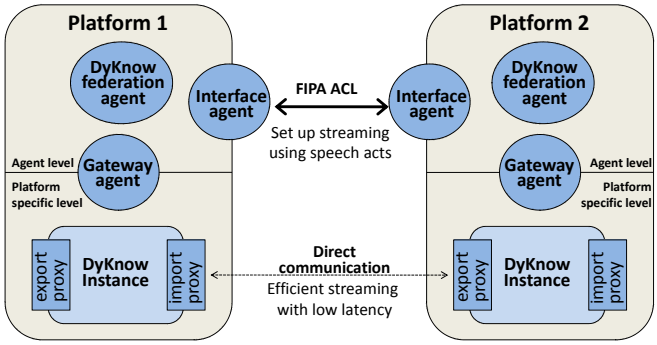

Figure 3. An overview of the components of a DyKnow Federation.

and the Gateway Agent is used to access the local DyKnow instance. Agents communicate using FIPA ACL while two DyKnow instances communicate directly through the Export and Import Proxies after setting up a stream through the DyKnow Federation service (Figure 3).

A DyKnow instance is made available to other platforms by integrating it in the agent framework in three steps:

1) By implementing the DyKnow Federation service,
2) by extending the Interface Agent to provide the DyKnow Federation service, and
3) by extending the Gateway Agent to allow the DyKnow Federation service to access the local DyKnow instance.

One important issue is how to refer to information among platforms. A DyKnow instance will contain a set of labeled stream generators. The easiest approach would be to use these labels directly. One problem with this approach is that the agent level must know what labels each of the other platforms use in their local DyKnow instances. This is not a major issue if all platforms are built by the same people, but in a more general setting this would not be easily done. A more feasible approach is to agree on a set of labels with a certain meaning among a group of agents called *semantic labels*. These semantic labels can then be translated by each agent to local DyKnow labels using whatever procedure necessary. For example, a group of UAVs could agree that the semantic label heli-position is used to refer to their own position. This is a first step towards introducing a common ontology. The benefits are that each group can use their own set of semantic labels, with a meaning they have agreed upon, while labels in the local DyKnow instances are isolated from each other.

1) The DyKnow Federation Service: The DyKnow Federation service is responsible for supporting the finding and sharing of information among local DyKnow instances. An agent which implements the DyKnow Federation service on a platform is called a DyKnow Federation Agent. The agent is registered in the local Directory Facilitator to make the platform available for federation. If an agent wishes to make a request to a DyKnow Federation Agent on a particular platform, it sends this request to the Interface Agent on that platform, which forwards the request to the DyKnow Federation Agent. The DyKnow Federation Agent is then responsible for fulfilling the request by using the Gateway Agent to access the local DyKnow instance.

The DyKnow Federation service supports the following:

1) Create a stream from a semantic label, a sample period, a maximum delay, a start time, an end time, and a set of receivers. This request should create a stream for the semantic label satisfying the given sample, delay, and duration constraint. This stream should then be exported to the receiver. For example, UAV A could send a create stream request for the semantic label heli-position to UAV B with a sample period of 2 seconds and UAV C as the receiver. If the request is accepted by UAV B, then it will export a stream with its helicopter position sampled every 2 seconds to UAV C.
2) Query the latest value, the value at a particular time-point, or all the values between two time-points for the stream associated with a semantic label. The answer should be returned to the sender in an inform message. For example, if a UAV would like to know the position of another UAV then it could query the other UAV about the latest value of the stream associated with the semantic label heli-position. If the query is accepted then the answer will be looked up in the local DyKnow instance and returned in an inform message.

2) The DyKnow Gateway: The DyKnow Gateway interface extends the Gateway Agent to allow the DyKnow Federation service to access the local DyKnow instance.

On the platform specific level there will be four CORBA servers, one for each of the interfaces (DyKnow Gateway, Export Proxy, and Import Proxy) and one for the DyKnow location making the imported streams available to the local DyKnow instance. The DyKnow Gateway will be called by the Gateway Agent, while the proxies are used on the platform specific level.

The interface the DyKnow Gateway should implement is:

- `create_stream`(s, f, t, p, d, u), where s is a semantic label, f, t, p, and d are the from, to, sample period, and delay arguments used to create a policy, and u is a set of receiving platforms.

The method should do the following:

1) Translate the semantic label s to a local label l;
2) create a policy p with the constraints from f, to t, sample period p, and delay d;
3) use the policy p and the label l to create a new stream in the DyKnow instance; and
4) export the stream by invoking the Export Proxy method `start_exporting`(l, s, w) for each receiver w in u.

3) Export Proxy: A component used by a platform to export one or more streams. To export a stream an internal subscription is made by the proxy which then makes the stream available to other platforms in an implementation specific way. It is also possible for the Export Proxy to use the DyKnow middleware to implement this functionality, but it might not be the best choice in all situations.

The interface that the Export Proxy should implement is:

- `start_exporting`(l, s, w), where l is a label, s is a semantic label, and w is a receiver.

The `start_exporting`(l, s, w) method creates a subscription to the stream generator l. Each time a new sample v is pushed on the stream the `push`(m, s, v) method is called on the Import Proxy object on the receiving unit w, where m is the unit number of the sending platform. The unit number of a platform is its unique identifier.

When implementing `start_exporting`, special care has to be taken since the other platform might no longer be available or it might be busy and not accept the call directly. One approach is to make the export proxy multi-threaded with one thread for each remote platform. In this way no other platforms will be affected by a stop in the communication.

4) Import Proxy: A component used by a platform to import one or more streams. Each imported stream will be provided as a source in the local DyKnow instance. How the stream is imported is an implementation detail which must be coordinated with the Export Proxy. Different pairs of proxies can use different methods to communicate.

The interface that an Import Proxy should implement is:

- `push`(m, s, v), where m is the sender, s is the semantic label, and v the sample.

The `push`(m, s, v) method translates the semantic label s to a label l and adds the sample v to the local stream generator associated with the label l. If this is the first sample for this semantic label then a new source is created and its stream generator is associated with the label l.

D. DyKnow Federation Functionalities

1) Adding and Removing Nodes: To make a node available for federation the Interface Agent of that node has to register its DyKnow Federation service in the Directory Facilitator. After this, the node is available, but no information is shared. To leave a DyKnow Federation, the DyKnow Federation service should be unregistered. Active streams to and from a node can be kept even after leaving the federation since the proxies talk directly to each other after the streaming has been set up.

2) Query for Information: If a platform needs a particular piece of information, knows its semantic label, and knows which platform can provide the information then a query can be sent to the Interface Agent of that platform with the semantic label as the argument. Using this method a platform could ask for the latest value of a stream, the value at a particular time-point, or all the values between two time-points.

If a platform knows the semantic label but not the platform then a global request can be made. The DyKnow Federation service will then query all platforms providing a DyKnow Federation service for the semantic label and return the result. If the service gets more than one answer then it has to either select one of the values or fuse them together.

If a platform does not know the semantic label it can make a request for all semantic labels which match an SL formula. SL is a first order content language developed by FIPA and used by JADE. To be able to write SL formulas an ontology must be created for the DyKnow Federation. This is done within the JADE framework by providing a concept for each semantic label. For example, if the ontology contains the concepts `Car`, `Color` and `OnRoad`, formulas using these can be written. To find all semantic labels of blue cars on road 7 the formula `(all ?x (and (Car ?x) (Color ?x blue) (OnRoad ?x road7)))` could be used. It is then up to each DyKnow Agent to interpret the formula and find all matching semantic labels. If a platform would like to know if any other platform has found the same car it is tracking then it could use this functionality to find potential matches.

To implement the first use case, explicit ask and tell to divide the monitoring area, this functionality would be used. The leader UAV would make a global request for the current value of the streams associated with semantic labels such as max-speed, fuel, and so on.

3) Streaming Information: Setting up a stream from one platform to another is different from requesting a particular piece of information directly. Instead of sending something back, the agent receiving the request will set up an Export Proxy which will start streaming the information to the Import Proxy of the requesting agent.

When proxies are set up the platform that made the request can access the stream through a local stream generator. From the point of view of the local DyKnow instance, the Import Proxy is another source of information, like a sensor.

This would be the main functionality required to implement the second use case, to provide continuous information about tracked vehicles from one UAV to another. The UAV receiving the stream would then have to fuse this stream with its own stream of car estimations in order to do the qualitative spatial reasoning and chronicle recognition.

V. The Traffic Monitoring Scenario Continued

We will now give some concrete examples of how the multi UAV traffic monitoring scenario introduced earlier can be implemented using the DyKnow Federation Framework.

The scenario starts with one of the UAVs, UAV A, being delegated the goal of monitoring an area for traffic violations from an operator. To collect information about available UAVs, UAV A sends a global request for the current value of the streams associated with the semantic labels heli-max-speed and heli-altitude. The actual message is:

```
(get-latest
  semantic-labels: (seq heli-max-speed
                        heli-altitude)
  to-units: (seq A))
```

Each UAV that receives the message through its DyKnow Federation Agent will return the latest value in the streams associated with the semantic labels. Assuming that only UAV B replies, UAV A divides the area between them according to their characteristics. UAV A then delegates the task of monitoring UAV Bs part of the area to UAV B. This constitutes a recursive delegation to UAV B of parts of the task delegated to UAV A. The two UAVs will then independently patrol their areas looking for traffic violations as described in [8].

Assume that UAV B detects the beginning of a potential traffic violation, in this case a partial match of a traffic violation chronicle. Further assume that one of the involved cars is

predicted to leave the area of UAV B before the end of the situation. To collect the information needed to determine whether there is a violation or not, UAV B requests a stream of car states for car5 sampled every 500 milliseconds from UAV A.

```
(create-stream
  semantic-labels: (seq car5-state)
  sample-period: 0.5
  to-units: (seq B))
```

When UAV A receives the request it has to anchor the symbol car5 internally. This can either be done by matching the symbol to an existing internal symbol, if such a symbol exists, or by collecting new sensor data from the object UAV B denotes by car5. In either case, UAV A needs more information about the symbol. This can for example be achieved by requesting the last 2 minutes worth of car states related to car5.

```
(get-trajectory
  semantic-labels: (seq car5-state)
  from: -120
  to-units: (seq A))
```

Using the received information UAV A can determine where car5 currently is and its identifying characteristics. Based on this UAV A can anchor the symbol to existing or new sensor data. More information about our anchoring approach can be found in [17]. As soon as UAV A is able to see an object that is anchored to car5, car states associated with the symbol will be exported to UAV B. This stream of car states will then be fused with local car states produced by UAV B, preferring local states if both are available. In this way, a continuous stream of observations of car5 is available and UAV B can correctly determine whether a traffic violation occurred or not, even though it did not itself observe the whole situation. It is important to note that nothing was changed in the local DyKnow instances of UAV A or B.

The benefit over other approaches is that the DyKnow Federation framework allows selective streaming of information with seamless integration in the local DyKnow application. What is streamed, by whom, and how often can be decided for each particular situation. The information can be on varying levels of abstraction depending on the task and the capabilities of the involved platforms. When the streaming has been set up, the receiving agent can treat the new stream as any other stream in its local processing providing seamless integration. Another distinguishing feature is that the specification of the streaming is negotiated within a high-level multi-agent framework while the actually streaming is carried out using efficient low-level protocols.

VI. Conclusions

A DyKnow Federation framework for collaborative information processing among UAVs has been presented. This type of framework is required to develop complex multi-agent systems where agents have to cooperate to solve problems which are beyond the capability of any individual agent. The framework allows agents to share and fuse information to provide more complete and accurate information.

The DyKnow Federation framework is an extension of the knowledge processing middleware framework DyKnow, integrating it with a FIPA compliant delegation framework. The extension allows an agent to share parts of its local DyKnow instance with other agents in a DyKnow Federation. The basic interaction and sharing is made on an agent level using the standardized FIPA ACL agent communication language. To increase the efficiency, direct communication is supported for continuous streaming of information between nodes. In both cases, the federation is used to find information and to set up the distribution.

Distributing and fusing information among multiple agents has been widely studied in many respects. This work provides a complete integrated system for knowledge processing both on the agent level and the multi-agent level. It shows how DyKnow can be extended to integrate not only sensing and reasoning on a single platform, but also sharing and fusing of information among multiple platforms.

In summary, we believe that the DyKnow Federation framework provides appropriate support for dynamically sharing and fusing information in a distributed network of platforms. Since the federation approach is very general and builds on a formal delegation framework it should be applicable to a wide range of complex multiple platform scenarios.

References

[1] P. Doherty, P. Haslum, F. Heintz, T. Merz, P. Nyblom, T. Persson, and B. Wingman, "A distributed architecture for autonomous unmanned aerial vehicle experimentation," in *Proc. DARS*, 2004.

[2] P. Doherty, "Advanced research with autonomous unmanned aerial vehicles," in *Proc. KR*, 2004.

[3] F. Heintz and P. Doherty, "DyKnow: An approach to middleware for knowledge processing," *J. of Intelligent and Fuzzy Syst.*, vol. 15, no. 1, 2004.

[4] F. Heintz, "DyKnow: A stream-based knowledge processing middleware framework," Ph.D. dissertation, Linköpings universitet, 2009.

[5] F. Heintz and P. Doherty, "A knowledge processing middleware framework and its relation to the JDL data fusion model," *J. of Intelligent and Fuzzy Syst.*, vol. 17, no. 4, 2006.

[6] F. White, "A model for data fusion," in *Proc. of National Symposium for Sensor Fusion*, vol. 2, 1988.

[7] J. Llinas, C. Bowman, G. Rogova, A. Steinberg, E. Waltz, and F. White, "Revisions and extensions to the JDL data fusion model II," in *Proc. Fusion*, 2004.

[8] F. Heintz, P. Rudol, and P. Doherty, "From images to traffic behavior – a UAV tracking and monitoring application," in *Proc. of Fusion*, 2007.

[9] P. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Comput. Surv.*, vol. 35, no. 2, 2003.

[10] F. Heintz, J. Kvarnström, and P. Doherty, "Bridging the sense-reasoning gap: DyKnow – stream-based middleware for knowledge processing," *Journal of Advanced Engineering Informatics*, vol. 24, no. 1, 2010.

[11] M. Ghallab, "On chronicles: Representation, on-line recognition and learning," in *Proc. KR*, 1996.

[12] D. Heimbigner and D. Mcleod, "A federated architecture for information management," *ACM Trans. Inf. Syst.*, vol. 3, no. 3, 1985.

[13] A. Sheth and J. Larson, "Federated database systems for managing distributed, heterogeneous, and autonomous databases," *ACM Comput. Surv.*, vol. 22, no. 3, 1990.

[14] P. Doherty and J.-J. C. Meyer, "Towards a delagation framework for aerial robotic mission scenarios," in *Proc. CIA*, 2007.

[15] FIPA, "Foundation for intelligent physical agents (FIPA) ACL message structure specification," http://www.fipa.org/, 2002.

[16] F. Bellifemine, G. Caire, and D. Greenwood, *Developing Multi-Agent Systems with JADE*. Wiley, 2007.

[17] F. Heintz, J. Kvarnström, and P. Doherty, "A stream-based hierarchical anchoring framework," in *Proc. of IROS*, 2009.