Multi-layer Data Acquisition for Rapid Data Analysis in Tactical Networks

Sharath Maligera Eswarappa

Lab Communication and Communicating Devices
Module: MA-INF 3304
WS19/20

University of Bonn, Institute of Computer Science 4
Endenicher Allee 19A, D-53115 Bonn, Germany

Lecturer: Prof. Dr. Peter Martini
Supervisors: Dr. Roberto Rigolin F. Lopes and Dr. Paulo H. Rettore
Student: Sharath Maligera Eswarappa
Matr.Nr: 3068883

Monday 15th June, 2020

Abstract

This document describes the conception and development of a real-time data acquisition system from multi-layer store and forward mechanisms implemented by TACTICS middleware. Our approach was motivated by the challenges of handling, cleaning and plotting data acquired from experiments conducted to model ever-changing communication scenarios in tactical networks. Development of this system began with an attempt to improve on labor-intensive manual methods of handling these challenges when experiments of longer duration were performed to an automated data acquisition system available, cutting down the time spent on summarizing the results of an experiment from days to minutes. Besides, with this system in place the number of users that can monitor real-time data is not limited.

Keywords - Data acquisition, Data analysis, tactical networks, ever-changing communication scenarios.

1 Introduction

Tactical communication systems typically consists of heterogeneous computing hosts and networking devices like Very High Frequency(VHF) and Ultra High Frequency(UHF) narrow band tactical radios operating in disruptive networking environments. These networking devices are characterized by extremely low bandwidth and require performance optimizations in order to support user-generated voice and data traffic. The tactical service infrastructure (TSI), introduced in [1] and developed in TACTICS project by various European partners enables tactical radio networks to participate in SOA (Service Oriented Architecture) infrastructures and provide as well as consume services to and from the tactical domain.

Tactical nodes are capable of running web services with SOAP (Simple Object Access Protocol). To avoid buffer overflow in the VHF radios when user-generated data exceeds buffer capacity, TSI middleware implements a store-and-forward mechanism with a hierarchical model of three queues complementing each other namely message, packet and radio buffer [2] as shown in Figure 1 left-side. In this figure, the hierarchical data pipeline starts from message queue ❶, followed by packet queue ❷ and finally to the radio buffer ❸; the cross-layer information exchange is facilitated by a contextual monitoring service (CMS) ❹ providing TSI node's current context information upon which queuing and admission decisions are made. The implemented Core Services are as follows: The *message queue* is used to decide the admission time between the layers based on contextual information. The *UDP transport* fragments the message into UDP packets. The *packet handler* transfers the packets into the radio buffer if the threshold is not reached. *Contextual monitoring* gathers the network topology and radio buffer occupancy details (using *radio plug-in*). Neighbor discovery is performed by Optimized Link State Routing (OLSRv2) protocol for neighbor discovery and compiles its findings into a routing table.

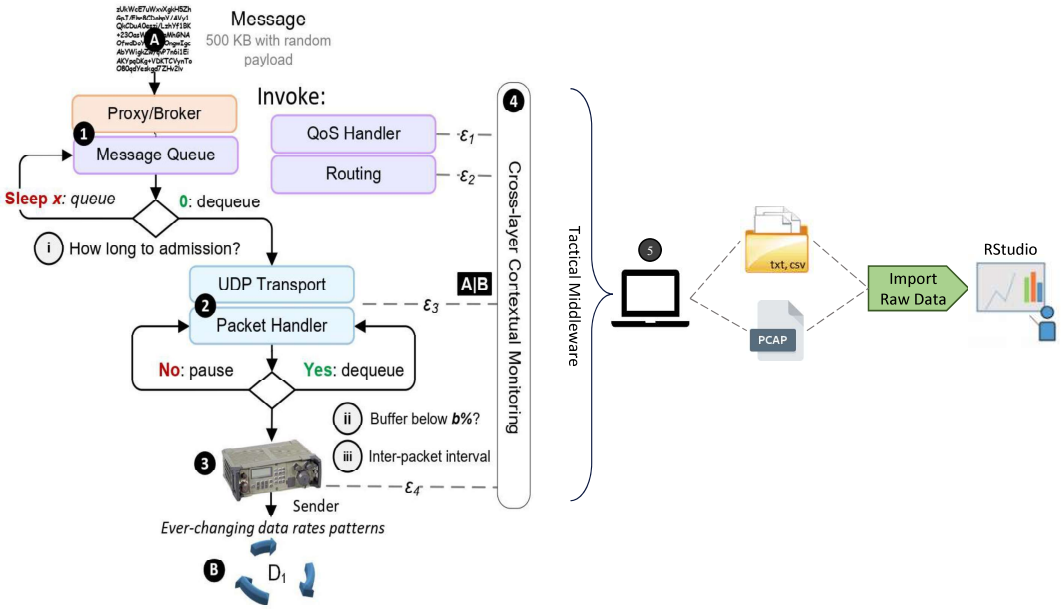


Figure 1: Previous state of Data Acquisition

To challenge the underlying store-and-forward mechanism, experiments performed to complement the investigations in [3] generate data from different layers of this queuing model. This data is then exchanged with the CMS. Figure 1 describes current approach of gathering data from CMS deployed on the tactical node ❺. On periodical intercepting of SOAP messages from CMS, we get real-time data with features from different layers of this model. The intercepted SOAP message is then written to a flat-file in XML document format. At the end of the experiment, each of these messages was then parsed using an XML parser to extract features from message queue, packet queue and radio buffer at different timestamps and written to corresponding three different flat-files. Along with these data, experiments conducted to characterize volatile and unpredictable behaviour of both user communication and network conditions generate network data in form of pcap files captured by tcpdump utility running on the tactical node ❺. Using this pcap file we need to manually derive the number of packets lost and network disruption intervals by comparing the timestamp at which packets are sent and received at tactical nodes.

With three flat-files and a pcap file in place we need to import and merge raw data from these files to form a common dataset file while removing irrelevant parts of the data. During post-processing of the dataset file, timestamp data having inconsistent format had to be manually corrected. New features regarding disconnection between radios and the number of packets lost in the process had to be inserted into this dataset file manually. After executing the data-cleaning process on these files, a statistical analysis and visualization tool like RStudio is used.

This process of manual data cleaning between flat-files, pcap file and the common dataset file requires considerable time and effort. Keeping this laborious approach in view the outcome of this lab course eliminates this painstaking manual data cleaning approach by proposing centralized real-time data acquisition platform to collect data from different layers of the queuing model, packet sniffer to capture network statistics both at sender and receiver VHF radios and the network condition between them. The extracted data are stored in a centralized database approach where data is synchronized based on timestamp improving data accuracy, consistency and accessibility.

The remainder of this paper is structured as follows: in next section, section 2, we discuss the background of tactical communication scenario, different layers representing functional blocks defined in NATO's C3 taxonomy and the need for this data acquisition approach. Section 3 describes our laboratory setup, method for deploying and provisioning of this system along with algorithms involved. In section 4 plots are shown summarizing the features acquired from the system. Finally, section 5 concludes the paper.

2 Background and Related Works

Tactical networks (TNs) are heterogeneous, mobile ad hoc networks (MANETs) hosting critical information systems for command and control (C2) in the battlefield. TNs are difficult and complex environments characterized by multiple restrictions that affect network behaviors, protocols, and systems. Tactical networking facilitates information sharing and data exchange among military tactical force units to enhance operational processes and situational awareness. It enables C2 system's capabilities for network-centric warfare [4]. These communication systems are typically operated in disruptive networking environments. The variations in network capabilities are dependent on factors such as operational mobility, terrain, communication media resources and network characteristics. Current networks support VHF/UHF line-of-sight (LOS) waveforms. Network constraints such as limited bandwidth and latency, narrow effective communication range, intermittent communication links and potential hidden nodes in certain operational terrains are the source of randomness changing network metrics such as link data rate, latency and packet loss. In addition, the mobility of tactical nodes causes rapid changes in network topology with the nodes leaving and joining the network on an ad hoc basis which demands multi-layer, distributed and self-configurable tactical systems not relying on centralized control.

2.1 Tactical network topology, node types and radios

Through the use of an already proven network infrastructure, it is necessary to consider the types of units deployed, their war-fighting platforms and their communication needs. In any given operation, there may be dismounted soldiers D1 , D2 , D3 , D4 , D5 as illustrated in Figure 2 operating in small units, forming a network of unpredictable mobility resulting in intermittent communication links 2 due to terrain masking. Shaping user generated data flow, D1 A dismounted soldier with handheld narrow band tactical radio (Starmille) operating at Ultra High Frequency (UHF) band supporting 240kBps over a range of 2kms shares positional information, text messages, etc., to the D5 dismounted commander at the tactical edge who then commands soldiers response to a changing tactical environment. The edge 3 connects the dismounted commander to the mobile node M1 having a tactical Very High Frequency (VHF) radio (9.6 kBps, 20 km) which then connects to the command post 6 over VHF or SatCom networks.

Regarding scalability of our testbed, observe that the UHF (Starmille) network can scale up to 14 radios (sharing 240 kBps), the VHF (PR4G) network up to 32 radios (sharing 9.6 kBps) and SatCom is point-to-point (500 kBps, emulated). Thus the most challenging scenario, using these particular military radios, would have 32 squads of 14 nodes each, summing up to 416 nodes in the tactical network (theoretically).

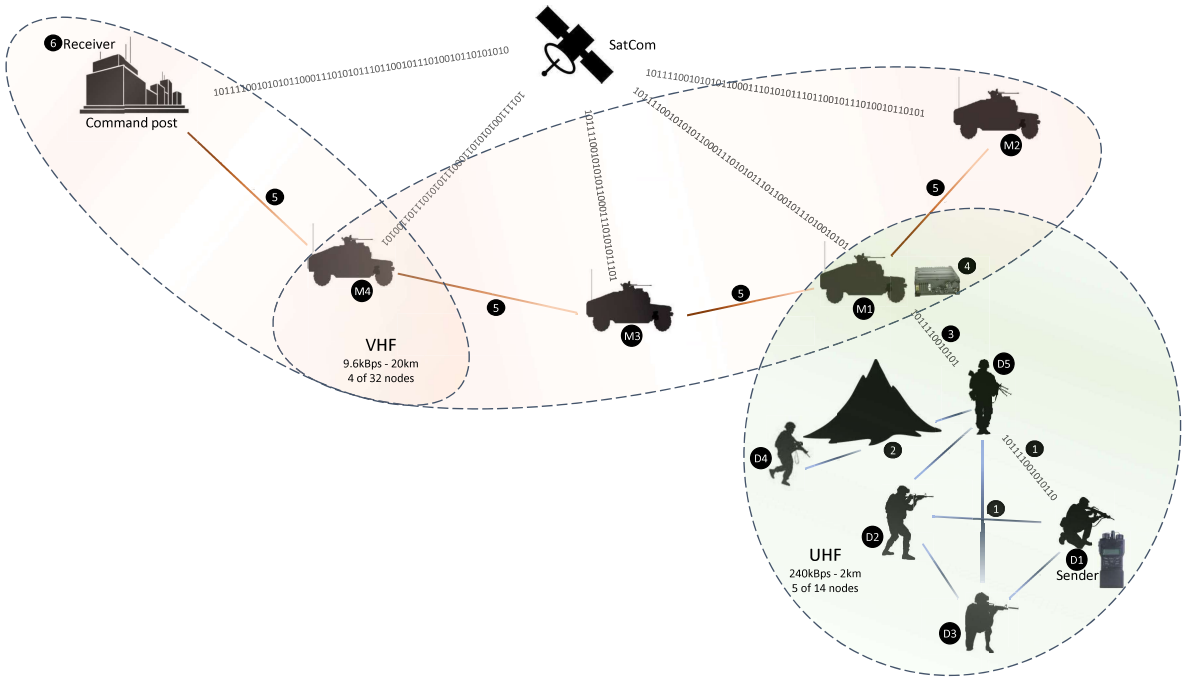


Figure 2: Network topology and node types

2.2 Multi-layer tactical systems

The C3 Taxonomy (Figure 3a) provides a generic framework [5] and contributes to a key component of the Connected Forces Initiative: Exploiting technology to help deliver inter-operability. *"Communication and Information System (CIS) Capabilities"* represents the logical components of the capabilities required to meet NATO's information system and communication needs in support of Missions and Operations. *Communication Systems* are systems or facilities for transferring data between persons and equipment. They usually consists of a collection of communication networks, transmission systems, relay stations, tributary stations and terminal equipment capable of interconnection and inter-operation so as to form an integrated whole. These individual components must serve a common purpose, be technically compatible, employ common procedures, respond to some form of control and generally operate in unison. *Information Systems* are integrated sets of components for collecting, storing, and processing data for delivering information, and digital products. Organizations and individuals rely on information systems to manage their operations, supply services and augment personal lives. *"User-Facing Capabilities"* provide an end user with *''User Applications"* which are computer software components designed to help a user perform singular or multiple related tasks running on *'User Appliances'* which provides the logical interface between human and automated activities. Applications are stable and relatively unchanging over time, whereas the services used to implement them will change over time, based on technologies and changing business needs. *"The Back-End Capabilities"* may be used to support the user-facing capabilities and are layered into Community of Interest (COI) Services, subdivided into COI-Specific Services and the more generic COI-Enabling Services, and the Core Services and Communication Services layers, both of which provide generic infrastructure capabilities.

Tactical systems are composed by at least four layers managing the ever-changing communication scenarios with dynamic user's data-flow A over ever-changing network conditions B. For example, Figure 3b depicts the in/out model with four layers representing the main functional blocks defined in NATO's Consultation, Command and Control taxonomy defined as follows:

(A) *User-facing services:* is a set of QoS-constrained command and control services available to the users such as conditions-of-interest, alerting and monitoring in tactical networks.

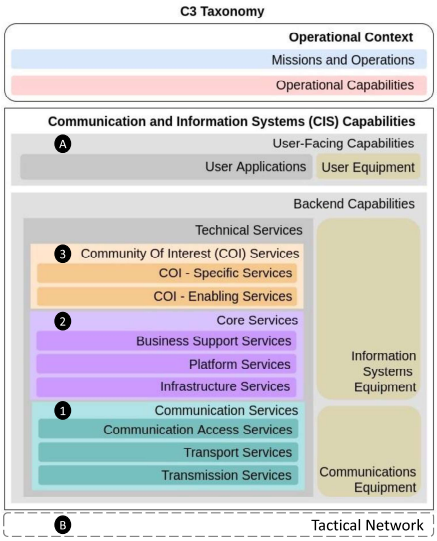
(3) *Community of Interest services:* hosts services to deal with particularities from the user applications(A) in a collaborative environment, such as coalitions among different nations.

(2) *Core Enterprise services:* provide generic, technical functionality to implement service-based environments for infrastructure, Service-Oriented Architecture (SOA) and enterprise support usually implemented within tactical systems.

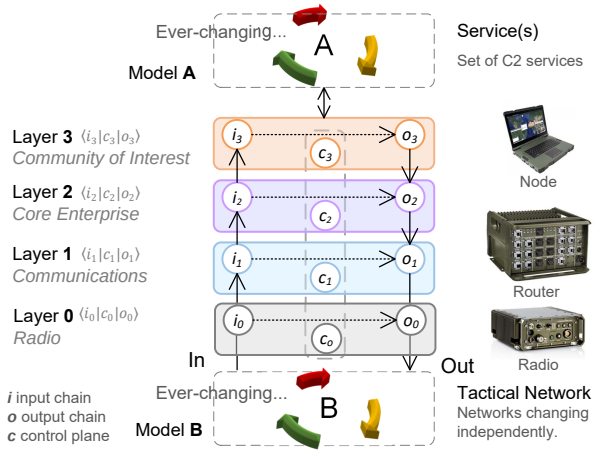
(1) *Communication services:* combine services for transmission, transport and network access, which are partially implemented by software-defined radios together with tactical routers managing the heterogeneous network.

(0) *Radio(s):* interface with the military radios composing the tactical network. This interface is usually implemented with standardized protocols like SNMP (Simple Network Management Protocol) and DLEP (Dynamic Link Exchange Protocol) to control/share network metrics like link data rate, radio buffer occupancy, latency, packet loss and so on;

(B) *Tactical Network:* heterogeneous radio networks with intermittent conditions and deployed in hostile environments, also potentially facing an adversary in both kinetic and digital domains.



(a) NATO's C3 taxonomy



(b) Layers with in/out chains [3]

Figure 3: Functional blocks from the services taxonomy (a) and layers with input/output data chains (b)

2.3 Related works

The Cross-layer Contextual Monitoring Service (CMS) collects cross-layer information from the TN's components and the radio(s), updates the metadata, and provides the context information to interested parties. Using flat-files (like CSVs), the data from CMS are inserted/appended to relevant flat-files. The data from these flat files are used in external programs like RStudio for statistical analysis. Data storage in these flat-file systems are adequate when the amount of data is small and one at a time read/write access is all that's needed. But when researchers need to perform experiments of longer duration maintaining and cleaning the resultant large amounts of data is painful and time consuming. When multiple stakeholders are impacted by the challenges imposed by store-and-forward mechanisms in tactical networks: Researchers, in need of agile methods to develop new tools and analysis algorithms require real-time data acquisition, storage and analysis approach. This is where re-directing the gathered data from CMS to a database system facilities the need allowing multiple user access to the database, writing/modifying data, running queries and performing any other tasks related to database management. Moreover RStudio provides Database interface (DBI) package defining communication between R and relational database management systems empowering us with rapid data analysis.

3 Multi-layer Data Acquisition Approach

Our laboratory setup is composed of four VHF radios ❶, each of which connected to a router ❷ and then to a laptop (node) ❸ as shown in Figure 4. This setup depicts multi-layer in/out models (Figure 3b) in tactical networks, with radios at layer 0, router at layer 1 and laptop at layer 2. The radio antennas were wired and connected to a coaxial relay ❺ which was then used to cut the connection between a pair of radios for a given time-window with the help of Raspberry Pi ❹ to mimic disruptive networking environment.

A centralized data acquisition approach was designed to support further investigations and experiments on tactical networks, avoiding previous semi-automated approach of data collection, handling and plotting the features collected from different layers of the queuing model and disconnection information. With this new approach, these features are written to a centralized well known relational PostgreSQL database hosted on ❼ as shown in Figure 4, providing continuous data query execution and analysis of these features in real-time. Working with a relational database in an agile approach can be tedious, repeatable process prone to human error which was the key motivation to deploy PostgreSQL database and automate provision of our data acquisition approach using an open source configuration management tool, Ansible; which is described in detail in section 3.1.

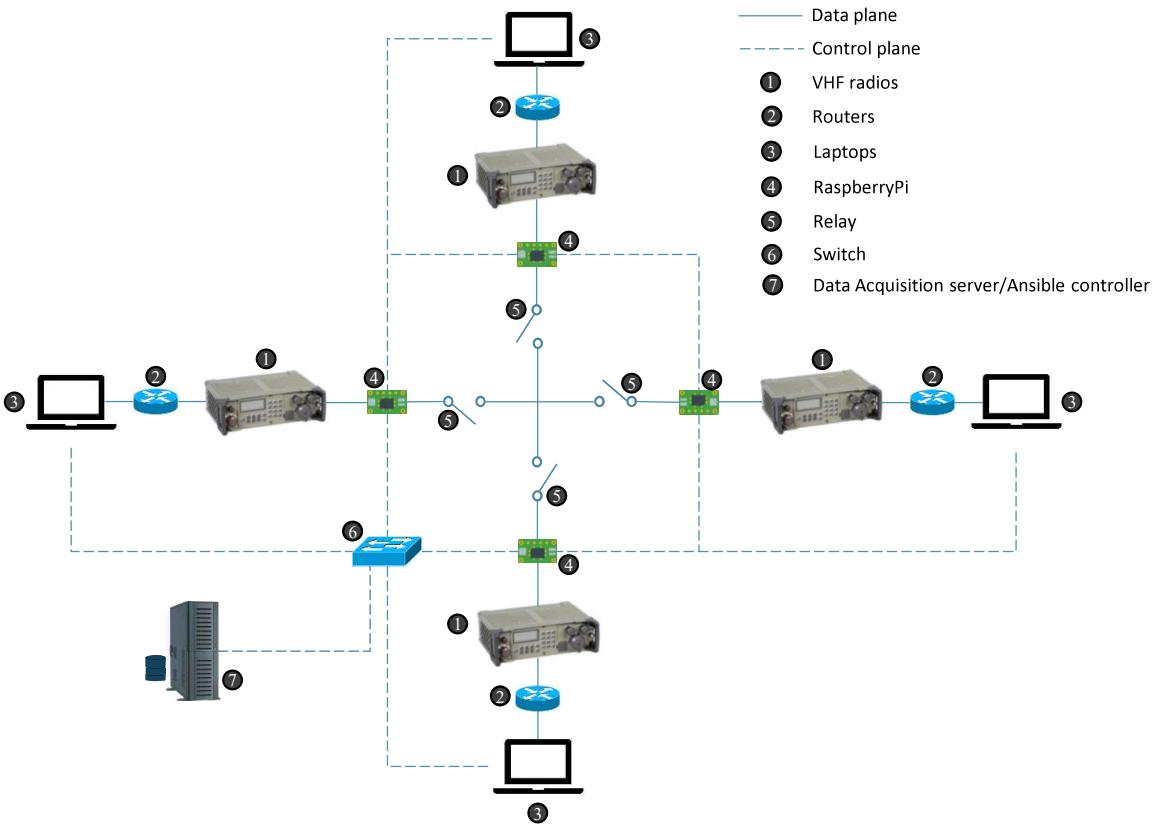


Figure 4: Experimental VHF network setup with four nodes (radios connected with wired antennas) connected to a management network together with a data acquisition server.

Figure 5 depicts the sequence of our new data acquisition process. We created synthetic data flow between a pair of VHF radios by sending a random message ❶ using SOUP UI tool to the sender node (Laptop1). The message is then added to the message queue ❷ where the message's priority and time-to-expire is used to sort out queue of messages and to drop/replace messages. The messages from the queue are then fragmented into IP packets at packet handler ❸ using UDP transport service. The packet handler service monitors radio buffer usage, enabling queuing at the queue of packets while the usage of the buffer is above a specific threshold, dequeuing when it is not.

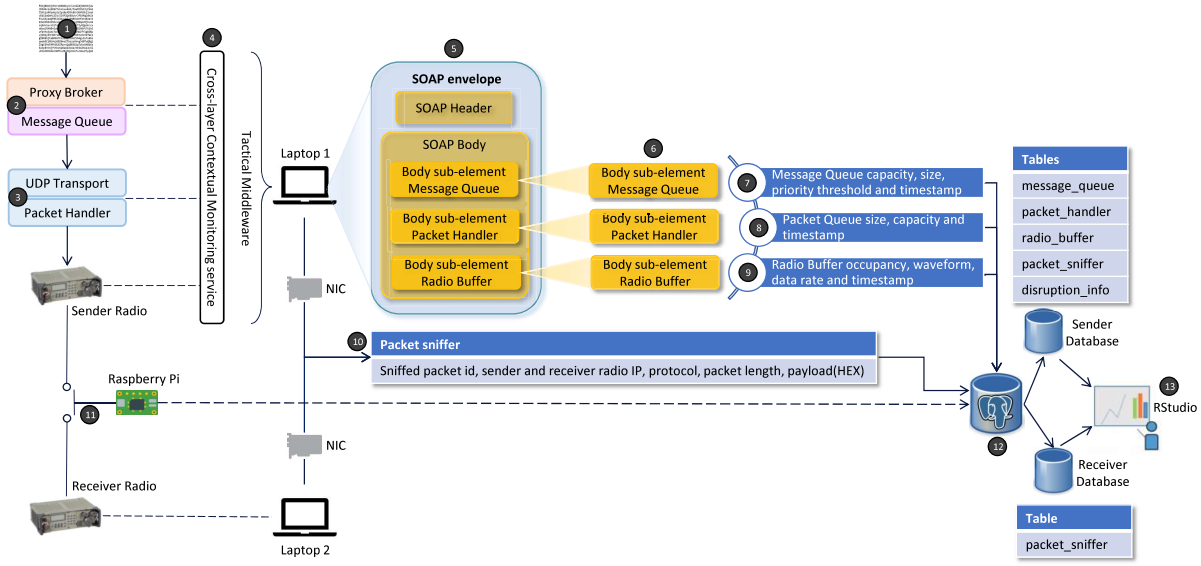


Figure 5: Data Acquisition Process

Using CMS ❹ we gather features from the queues and radio buffer in real-time. CMS response of a request consists of a SOAP envelope ❺ which is the root element containing two child elements, an optional header element and a mandatory Body element. The Body element consists of information intended for the recipient of the message. In our case it consists of three sub-elements ❻ for message queue, packet queue and radio buffer. Each time we get a response from CMS, we parse this SOAP envelope to get those sub-elements using a python script. The algorithm employed in parsing the envelope and extracting individual sub-elements and writing the extracted features from these sub-elements to the database server ⓬ is described in Algorithm 1. Let us explain this algorithm in detail, the procedure AcquireData (line 1) describes parsing of the SOAP envelope using python BeautifulSoup package to find Body element and timestamp value at which statistics of our queuing model were signed and sent as part of the message. Body element is then passed on-to the three different asynchronously executing procedures WriteRadioBufferData (line 7), WriteMessageQueueData (line 8) and WritePacketHandlerData (line 9) to write the features extracted from radio buffer, message queue and packet handler respectively to the 'sender database' hosted on database server ⓬.

In procedure WriteRadioBufferData (line 14) we find radio buffer sub-element node from the Body element and then enumerate on the node's tag and it's attribute as corresponding key-value pairs. Keys represent radio_buffer table columns as defined in the Entity-Relationship diagram in Figure 6. The corresponding values of the keys were casted to the respective data type and then inserted as a record in the radio_buffer table using procedure SaveDataToDb defined in Algorithm 4. The columns in this table represent radio buffer occupancy in Bytes, type of waveform and data throughput of the link in kiloBytes per second acquired at the current timestamp as depicted in ❾ of Figure 5. Similarly, in procedures WriteMessageQueueData (line 25) and WritePacketHandlerData (line 36) we parse the Body element to find message queue and packet handler nodes respectively. Upon finding these nodes, we iterate over their tags and attributes to insert data to the message_queue and packet_handler tables as defined in Figure 6. The columns in message_queue table corresponds to the queue capacity, it's size in Bytes and message's priority at current timestamp ❼. The columns in packet_handler table corresponds to packet queue size, it's occupancy in Bytes at current timestamp ❽.

In order to capture network metrics such as link data rate and packets lost during disconnection between sender and receiver We have a procedure called PacketSniffer defined in Algorithm 2. We use Berkeley Packet Filter applied on the network interfaces of Laptop 1 and Laptop 2 as shown in Figure 5. While executing this filter using python pyshark package, we sniff continuously for UDP packets at these interfaces ❿. On capture of UDP packets at both sender and receiver nodes we extract UDP packet information such as packet id, source and destination ip address, packet length and payload in HEX string format. We write these features to the packet_sniffer tables defined as in Figure 6 to both sender and receiver databases based on which node's interface the packets were sniffed.

Algorithm 1 Data Acquisition from TSI nodes

Input: Target database 'to_db', Target schema 'to_schema', time interval 'T'
Output: Populated Database P_D

1: **procedure** AcquireData(to_db, to_schema)
2: **while** True **do**
3: sleep(T)
4: envelope element, E_S← get CMS response as a SOAP message
5: body element, B_S← parse E_S to find it's body using BeautifulSoup package
6: timestamp, T_S← parse B_S to find timestamp of CMS response and convert it into ISO datetime format
7: WriteRadioBufferData($B_S, to\_db, to\_schema, T_S$)
8: WriteMessageQueueData($B_S, to\_db, to\_schema, T_S$)
9: WritePacketHandlerData($B_S, to\_db, to\_schema, T_S$)
10: **end while**
11: **return** P_D
12: **end procedure**
13:
14: **procedure** WriteRadioBufferData($B_S, to\_db, to\_schema, T_S$)
15: **Input**: radio_buffer table columns
16: **Output**: Populated radio_buffer table R_T in to_db, to_schema
17: **Begin**
18: N_R← parse B_S to find radio buffer node in CMS response
19: D_R← parse N_R to find tag and it's attribute as key-value pair w.r.t radio buffer occupancy, waveform and data rate
20: SaveDataToDb($D_R, to\_db, to\_schema, T_S$)
21: **End**
22: **return** R_T
23: **end procedure**
24:
25: **procedure** WriteMessageQueueData($B_S, to\_db, to\_schema, T_S$)
26: **Input**: message_queue table columns
27: **Output**: Populated message_queue table M_T in to_db, to_schema
28: **Begin**
29: N_M← parse B_S to find message queue node in CMS response
30: D_M← parse N_M to find tag and it's attribute as key-value pair w.r.t message queue capacity, size at T_S and message priority
31: SaveDataToDb($D_M, to\_db, to\_schema, T_S$)
32: **End**
33: **return** M_T
34: **end procedure**
35:
36: **procedure** WritePacketHandlerData($B_S, to\_db, to\_schema, T_S$)
37: **Input**: packet_handler table columns
38: **Output**: Populated packet_handler table P_T in to_db, to_schema
39: **Begin**
40: N_P← parse B_S to find packet handler node in CMS response
41: D_P← parse N_P to find tag and it's attribute as key-value pair w.r.t packet queue size and occupancy at T_S
42: SaveDataToDb($D_P, to\_db, to\_schema, T_S$)
43: **End**
44: **return** P_T
45: **end procedure**

To emulate disruptive networking environment we use prototype device to disconnect the coaxial cable using a Raspberry Pi and a relay (see ⓫ in Figure 5) between the sender and receiver radio. Algorithm 3 describes the algorithm employed to disconnect and save time intervals at which we initiate the disconnection. In procedure SetUpRaspberryPi (line 1) we set up Raspberry Pi's General Purpose Input

Algorithm 2 Sniff packets to capture network statistics at tactical nodes

Input: *Berkeley Packet Filter 'bpf'* executing the filter program, 'to_db', 'to_schema', packet_sniffer table columns
Output: Populated table S_T in to_db, to_schema

1: **procedure** PacketSniffer(bpf, to_db, to_schema)
2: **while** True **do**
3: packet← sniff packets continuously from the interface using *bpf* and pyshark package
4: packet_data← get packet id, sniffed timestamp, source ip, destination ip, protocol, packet length, payload length and payload in HEX string format
5: SaveDataToDb($packet\_data, to\_db, to\_schema, T_S$)
6: **end while**
7: **return** S_T
8: **end procedure**

Output (GPIO) pins, it's ip address and port number on which it listens to the disconnection instruction. In procedure InitDisruption (line 11) we implement disruption between the nodes by informing the Raspberry Pi to switch on the relay when disconnection instruction is received from either of the two nodes. We simultaneously log the timestamp of connection and disconnection states of the link to disruption_info table as defined in Figure 6. If there is a disconnection between nodes, log as boolean value 'TRUE' on the column is_disrupted, 'FALSE' otherwise.

Algorithm 3 Save disruption information between two tactical nodes i.e between sender and receiver VHF radios

Input: to_db, to_schema, disruption interval T_D in seconds
Output: Populated disruption_info table D_T in to_db, to_schema

1: **procedure** SetUpRaspberryPi
2: **Input**: Raspberry Pi ip address and port number to listen to
3: **Output**: Socket object S_O
4: **Begin**
5: Set up Raspberry Pi General Purpose Input Output pins using BCM numbering system
6: Create socket object S_O and listen to disruption instruction on it
7: **End**
8: **return** S_O
9: **end procedure**
10:
11: **procedure** InitDisruption($T_D, S_O, to\_db, to\_schema$)
12: **Input**: Raspberry Pi ip address and port number to listen to
13: **Output**: Socket object S_O
14: **while** True **do** wait for sender radio to get connected to S_O
15: **if** Connected established and disruption instruction received from sender radio **then**
16: switch on the relay between sender and receiver radio
17: sleep(T_D)
18: switch off the relay
19: log the timestamp at which the disruption instruction received as TRUE otherwise FALSE in the table D_T in to_db, to_schema
20: **end if**
21: **end while**
22: **return** D_T
23: **end procedure**

To insert all the data acquired using aforementioned approaches to the database server we use psycopg2 PostgreSQL database adapter for python. Psycopg2 was designed for heavily multi-threaded applications that can create and destroy lots of cursors and make a large number of concurrent INSERT's or UPDATE's. In Algorithm 4 we have a procedure called SaveDataToDb (line 1) where we define target database and schema for the data to be inserted. Once a connection has been established between python process acquiring data to the target PostgreSQL database using psycopg2, we first check whether the connection is being established for the first time during the initial phase of the experiment, if yes we

send an alert to the user that the corresponding database tables, where data has to be inserted, will be truncated. If user choose to proceed, the tables will be truncated and records will be inserted else the experiment will be aborted.

Algorithm 4 Insert Data to PostgreSQL database

Input: Target database 'to_db', it's ip address, username and password, Target schema 'to_schema'
Output: Populated table T in to_db, to_schema

1: **procedure** SaveDataToDb(D, to_db, to_schema, T_S)
2: **Begin**
3: connection← get connection to the target database using psycopg2 python PostgreSQL database adapter
4: cursor← get cursor object from connection
5: Truncate table T if it exists in to_db, to_schema and if it is the first time connecting to the database
6: query← construct PostgreSQL query using keys of D as columns of the table and corresponding value as rows that has to be inserted
7: execute query on cursor object
8: commit the connection
9: close the connection
10: **End**
11: **return** T
12: **end procedure**

After concluding our experiment, database tables message_queue, packet_handler, radio_buffer, packet_sniffer and disruption_info (tables in Figure 6) is populated. We now have to gather all the data from these tables in a common table so that at any discrete timestamp we have all the features we need to do statistical analysis on our experiment. To do this, we developed the Algorithm 5 where procedure GenerateTimeSeries (line 1) generates time series with an interval of one second between minimum and maximum timestamp considering timestamp values from all the aforementioned tables. Using these time series values we perform inner join on these tables in procedure GenerateFeatureDataset (line 8) to form a new table feature_dataset (see Figure 6) combining all the features at one place. Using RStudio's DBI package for PostgreSQL database, we connect to this feature_dataset table (⓭ in Figure 5) from R and get all the columns as a data frame using which we plot figures for statistical analysis of our experiment.

Algorithm 5 Generate feature dataset combining all features at one place

Input: Database tables radio_buffer R_T, message_queue M_T, packet_handler P_T, packet_sniffer S_T and disruption_info D_T
Output: Populated feature dataset table F_T

1: **procedure** GenerateTimeSeries(R_T, M_T, P_T, S_T, D_T)
2: minimum timestamp, T_{min}← select minimum timestamp accross all tables R_T, M_T, P_T and S_T
3: maximum timestamp, T_{max}← select maximum timestamp accross all tables R_T, M_T, P_T and S_T
4: T_{array}← generate a series of timestamp between T_{min} and T_{max} with one second interval
5: **return** T_{array}
6: **end procedure**
7:
8: **procedure** GenerateFeatureDataset($R_T, M_T, P_T, S_T, D_T, T_{array}$)
9: **for** each element T_n in T_{array} **do**
10: tuple ← find tuple in R_T, M_T, P_T, S_T and D_T
11: **if** tuple is found **then**
12: combine all the column features of R_T, M_T, P_T, S_T and D_T to populate feature_dataset table F_T
13: **end if**
14: **end for**
15: **return** F_T
16: **end procedure**

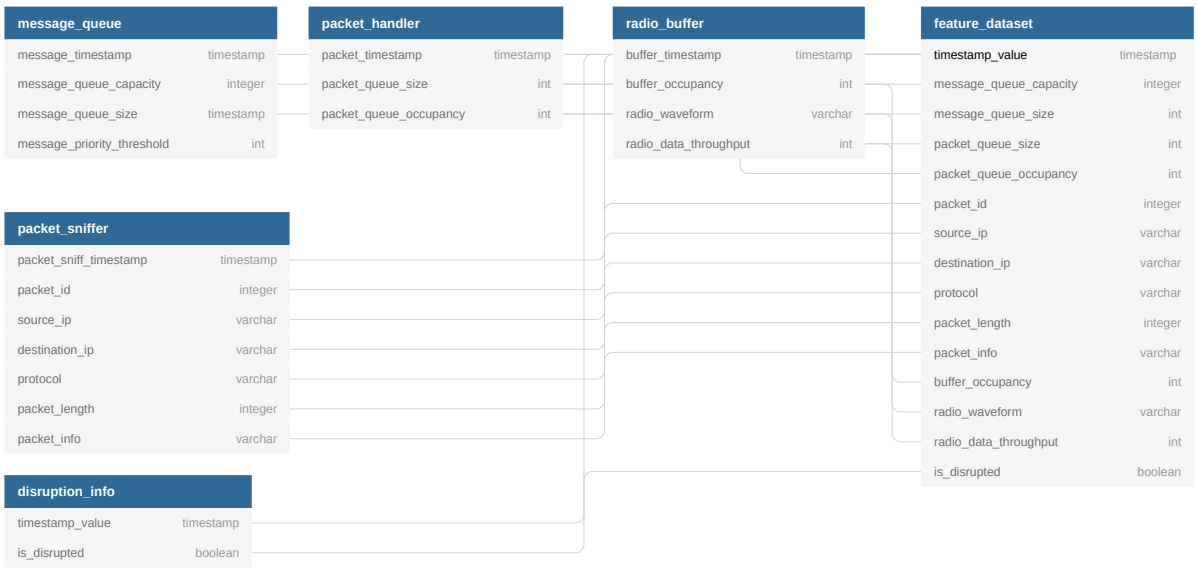


Figure 6: Entity-Relationship Diagram

3.1 Automated Provisioning of Data Acquisition

Infrastructure As Code (IaC) is an emerging practice to describe and specify the underlying infrastructure of software systems and their configuration. IaC scripts helps us to manage configuration and infrastructure automatically. Idempotency is a principle of IaC – it is the property that a deployment command always sets the target environment into the same configuration, regardless of the environment's starting state. Idempotency is achieved by either automatically configuring an existing target or by discarding the existing target and recreating a fresh environment.

Configuration Management (CM) is a process of ensuring consistency in infrastructure. Chef, Puppet, Ansible, and SaltStack are all 'configuration management' tools, which means they are designed to install and manage software on existing servers. The previous deployment techniques make use of batch scripts which lack a lot of features including idempotency and is considered as an inadvisable approach. The goal is to automate the whole process of deploying using a configuration management tool. For this purpose, in our case, the Ansible tool which was developed to simplify complex orchestration and configuration management tasks has been chosen.

The Ansible framework itself has been written in Python and allows users to script commands in YAML Ain't Markup Language (YAML) as an imperative programming paradigm. The main reason for using Ansible as our configuration management tool is the fact that it is designed for multi-tier deployment. The architecture of Ansible is agentless meaning we don't have to install any client-side software. It works purely on SSH connections. This also means that we can install it only on one system and control our entire infrastructure from that machine. To install Ansible server application, a system with a Unix or Linux operating system that contains Python 2.6 or greater is necessary. The installation itself was done from a Python's package management tool (pip). Ansible doesn't require any database and doesn't need any daemons running. This makes it easier to maintain versions and upgrades without any breaks. In our set-up (see Figure 7), we call the machine where we install Ansible as ***"controller"***. To establish communication between TACTICS nodes, a pair of ssh keys (private and public) is required. Ansible controller has a pair of these keys, which represents the identity of the given server and allows the initialization of encrypted connection. The ssh-keygen command was used to generate these keys.

By default, Ansible let's us manage our inventory in a text file called 'hosts' which is considered to be a single point of truth. Ansible can run its tasks against multiple hosts in parallel. For such parallel execution, it allows us to group our hosts in the 'hosts' inventory file and take all the hosts that fall under that group and executes tasks against those. In our set-up as shown in Figure 7 there are three groups with four tactical nodes (radio1 to radio4), four raspberry pi's (pi1 to pi4) and a database server. Each machine can also be a member of several groups. We have an Ansible.cnf configuartion file where we define settings in, like environment variables, command-line options, playbook keywords and variables.

We have a playbook which is a yml file containing one or more plays mapping group of hosts to a well defined tasks. Plays also define the order in which tasks has to be executed against those group of hosts. In our set-up, data acquisition platform will be provisioned only after PostgreSQL database is deployed.

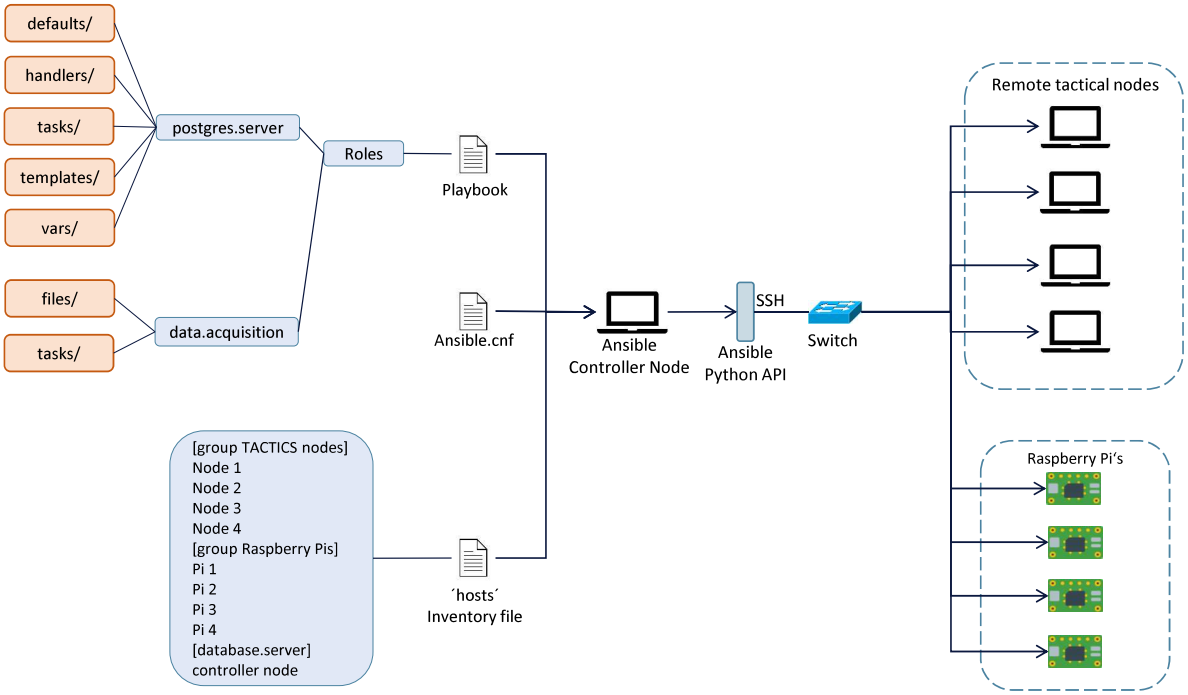


Figure 7: Laboratory Ansible architecture

Writing tasks in a single playbook may work fine for a very simple setup. However, if we have multiple services spanning across number of hosts, this will quickly become unmanageable. This is where Ansible roles are beneficial, allowing us to configure groups of hosts at the same time, avoiding repeat the entire process. Using a role we can encapsulate a group of meaningfully related tasks. Each playbook contains one or more roles that provision one or more hosts by executing tasks. In our case, we are deploying postgreSQL database using ***'postgres.server'*** role and provisioning data acquisition platform using ***'data.acquisition'*** role. In next sections 3.1.1 and 3.1.2 we describe the tasks accomplished by these roles.

3.1.1 postgres.server role

'postgres.server' role installs and configures PostgreSQL database server on Ubuntu server (see ❼ in Figure 4). As shown in Figure 7, postgres.server role directory contains defaults, handlers, tasks, templates and vars sub-directories. Each of these sub-directories except templates typically contain a main.yml file, which is the default file specifying the sequence of operations to be performed.

User-defined variables for PostgreSQL database version, it's data directory, configuration directory and installation directory are specified in Vars/main.yml which will be used throughout this role. The defaults directory is designated for default variables like unix socket directories, global configuration options, port number etc., that take the lowest precedence. In other words, If a variable is defined nowhere else, the definition given in defaults/main.yml will be used. Templates allow us to dynamically construct our playbook and its related data such as variables and other data files. Ansible uses the Jinja2 templating language to pass configuration variables to templates during run time creating a text file, which is then copied to the destination host.

When we change a PostgreSQL database configuration, we need to restart the 'postgresql' service so that it reads our modifications and applies those. This is where handlers come in. If we launch Ansible after making configuration change, while executing, it will compare the file inside our role with the one on the system, detect the configuration drift and copy it over to the changed file. In our case, we will

call a handler in handlers/main.yml to restart the 'postgresql' service after sleeping for 5 seconds on configuration change. Overall with *'postgres.server'* role, we accomplish the following tasks :

1. Ensure PostgreSQL packages and Python libraries are installed.
2. Ensure all configured locales are present and force-restart PostgreSQL after new locales are generated.
3. Set PostgreSQL environment variables and ensure PostgreSQL data and unix socket directories exist.
4. Ensure PostgreSQL database is initialized.
5. Configure host based authentication (if entries are configured) and restart 'postgresql' service.
6. Ensure that PostgreSQL databases for each tactical node are created.
7. Create schemas if not exists with respect to different experiments in each of those databases.
8. Create all the required tables if not exist in each of those schemas.
9. Ensure all postgres database users are present and grant ALL privilages to them.
10. Ensure postgres database user has access to all tables, sequences in all schemas, in all databases.

3.1.2 data.acquisition role

'data.acquisition' role provisions our data acquisition platform. The data.acquisition role directory, see Figure 7, contains tasks and files sub-directories. The task directory includes all the tasks in tasks/-main.yml that this role will run. Typically, the files directory is used to copy over static files to the destination hosts. In our case, we copy python scripts for data acquisition, packet sniffer, raspberry pi disruption and feature dataset generation to remote tactical nodes and raspberry pi devices. With *'data.acquisition'* role, we accomplish following tasks on remote tactical nodes and raspberry pi's:

1. Installing essential packages like pip, python-dev, libffi-dev, libssl-dev, locales, build-essential, ntp, htop, libpq-dev, python3-psycopg2, tshark, Logbook, beautifulsoup4, lxml, pyshark, numpy.
2. Install Python 3.6 along with necessary packages to install it.
3. Create a folder(if not exists) in usr/share on remote hosts and copy the scripts in files directory to it.

4 Initial results

We conducted experiments which supported our investigations in [6], [7] on the laboratory setup as illustrated in Figure 4 between two nodes where one node act as user message sender radio and other acting as message receiver radio. We use these VHF radios to simulate the adaption of tactical systems to ever-changing *user behaviour* and *network conditions* as described below.

User behaviour: To reproduce the outburst of user-generated messages in response to situations of conflict and requests of information in battlefield at tactical edge (D5 in Figure 2), we have used the SOAP-UI (Simple Object Access Protocol - User Interface) tool to generate a 500kB message. The maximum capacity of the radio buffer in our set-up is 128 kB, in which case the generated message is 4 times larger than the buffer. One can be sure that such a huge message can cause buffer overflow in the absence of any control mechanism.

Network Conditions: We have chosen the data rate as the network metric to replicate varying network conditions because data rate has direct correlation with the data transfer, for example: higher the data rate, closer the nodes are and more number of packets are sent/received in the network and vice-versa. We model this ever-changing data rate by a stochastic model using Markov chains[3] to define a probability distribution of creating different patterns of data rate change. This model simulates randomness in the

links resulting in intermittent communication (❷ in Figure 2) due to mobility and physical obstacles. The output of the data acquisition process results in the extraction of features from different layers of the queuing model (Figure 1) and link data rates between sender and receiver radio. For this purpose a database will be created for both sender and receiver radio on the data acquisition server (❼ in Figure 4) populated with relevant features and disconnection information. Figure 6 describes Entity-Relationship Diagram between the tables created on schema corresponding to experiments conducted on radios.

Figure 8 shows the features acquired using our proposed data acquisition approach from an experiment and plotted using R Studio's DBI package for PostgreSQL database. The experiment was conducted by sending 500kB using SOAP-UI and monitoring packet layer and radio buffer at sender radio. In message queue we measure the queue capacity which was constant throughout our experiment. The occupancy of packet queue is measured in KiloBytes(kB). Radio buffer occupancy determines the percentage of the total capacity of the buffer occupied by packets. For example, if the radio buffer occupancy is 10, then it means that radio buffer is filled by 10% of 128kB of data. The six supported link data rates between VHF radios has been color categorized into *Disconnected(red)* $\sim$ 0kBps, *Limited(yellow)* $\sim$ {0.6, 1.2, 2,4}kBps and *Connected(green)* $\sim$ {4,8, 9.6}kBps. We simulated varying link data rates between the radios using radio's SNMP interface (Simple Network Management Protocol) along with two disconnections for 60 seconds each. Using these plots we can infer that when the link quality has considerably higher data rate (*Connected*), the packet transmission rate increases thereby indicating lower packet queue and radio buffer occupancy. We can also calculate the number of instances when the radio buffer occupancy is above and below the threshold in a total of t seconds along average buffer occupancy for every t seconds.

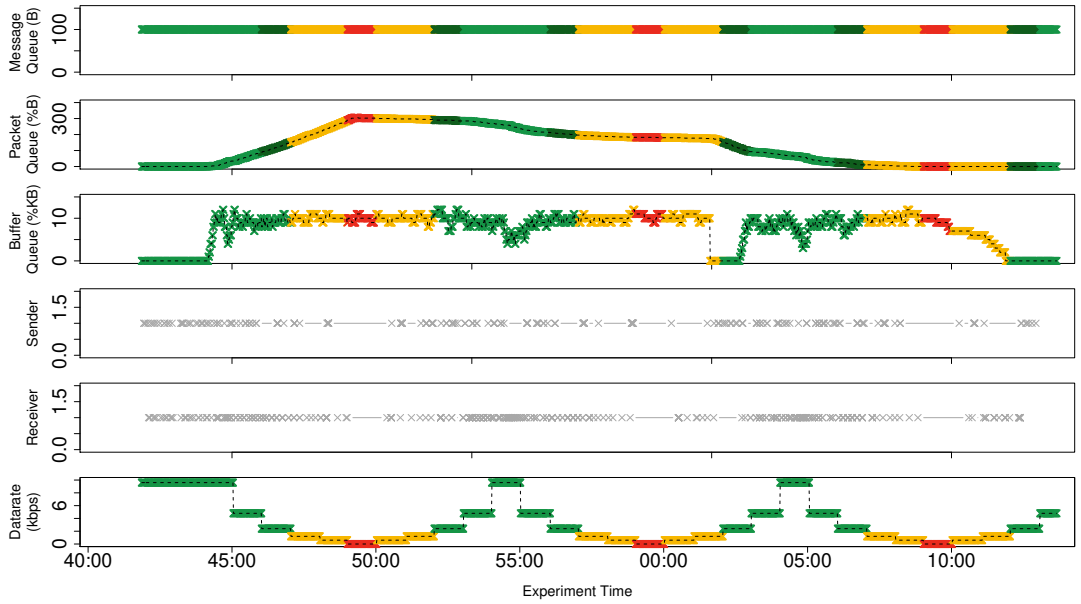


Figure 8: Features from queuing model during dynamic link data rates between radios.

5 Conclusion

In this report we have described in detail our novel data acquisition platform that collects all the features from multi-layer store and forward mechanism implemented in TACTICS middleware and writing it into a centralized PostgreSQL database server in real-time. Along with these features we could also collect network metrics like link data rate, connection/disconnection status between a pair of VHF radios. We also keep track of packets that are sent and received at sender and receiver radio respectively so that in case of loss of packets during disconnection we could re-transmit those. All these collected features are stored and sorted based on timestamp with respect to each tactical node in the centralized database server where every node can access information of other nodes.

We have also explained in detail our data acquisition approach starting from our laboratory VHF network

setup to algorithms used for parsing intercepted SOAP message from cross-layer CMS while extracting features related to our hierarchical queuing model in the process. Algorithms have been defined to sniff packets on tactical nodes to capture network statistics, to instruct and save connection/disconnection status between two tactical nodes. After capturing features using these algorithms we have explained how we insert data to corresponding tables along with modelling of these data for use with databases.

We have described the steps involved in deploying and provisioning of our platform using Ansible configuration management tool. We have shown plots summarizing the results of an experiment done using this platform, proving how useful it is in arriving conclusion of an experiment by reducing time and effort. Also with this platform in place, multiple users can access and monitor data simultaneously for their individual statistical analysis.

Further, we are working on improving this platform by introducing Apache Spark unified analytic engine for large-scale data processing that can work on both batch and real-time analytics in a faster and easier way. Apache Spark provides Java Database Connectivity (JDBC) driver to PostgreSQL database can retrieve current records from database tables as Data Frames. Once these Data Frames are created, we can apply various actions like applying a machine learning model to estimate link quality and predict disconnection between radios. Simultaneously, we are also working on building this machine learning model with a hypothesis that, given features from the hierarchical queuing model (message queue, packet and radio buffer) and network metrics between radios the model can learn to apply aforementioned actions. Every next level with this data acquisition platform, we would like to improve the way we acquire, analyze and act upon the data from tactical systems testing their performance bounds.

References

[1] A. Diefenbach, T. Ginzler, S. McLaughlin, J. Śliwa, T. A. Lampe, and C. Prasse, "Tactics tsi architecture: A european reference architecture for tactical soa," in *2016 International Conference on Military Communications and Information Systems (ICMCIS)*. IEEE, 2016, pp. 1–8.

[2] R. R. F. Lopes, A. Viidanoja, M. Lhotellier, A. Diefenbach, N. Jansen, and T. Ginzler, "A queuing mechanism for delivering qos-constrained web services in tactical networks," in *2018 International Conference on Military Communications and Information Systems (ICMCIS)*. IEEE, 2018, pp. 1–8.

[3] R. R. F. Lopes, P. H. Balaraju, and P. Sevenich, "Creating ever-changing qos-constrained dataflows in tactical networks: An exploratory study," in *2019 International Conference on Military Communications and Information Systems (ICMCIS)*. IEEE, 2019, pp. 1–8.

[4] D. S. Alberts, J. J. Garstka, and F. P. Stein, "Network centric warfare: Developing and leveraging information superiority." Assistant Secretary of Defense (C3I/Command Control Research Program ..., Tech. Rep., 2000.

[5] J. E. Hannay, "Architectural work for modeling and simulation combining the nato architecture framework and c3 taxonomy," *The Journal of Defense Modeling and Simulation*, vol. 14, no. 2, pp. 139–158, 2017.

[6] R. R. F. Lopes, P. H. Balaraju, P. H. Rettore, S. M. Eswarappa, J. Loevenich, and P. Sevenich, "Quantifying the robustness of tactical systems over ever-changing data rates," in *32nd International Teletraffic Congress (ITC 32)*, Osaka, Japan, Sep 2020, to appear.

[7] P. H. Balaraju, P. H. Rettore, R. R. F. Lopes, S. M. Eswarappa, and J. Loevenich, "Dynamic adaptation of data flow to evaluate the robustness of a tactical system," in *11th International Conference on Networks of the Future (NoF)*, Bordeaux, France, October 2020, to appear.