Real-Time Electronic Voting System

Marchand, Frederic	McBride, Tyler	Schurman, Brandon	Sikiru, Ibrahim
100817579	100888344	100857068	100887589

March 20, 2015

In this project, we build an electronic version of the Canadian electoral system which is based on a parliamentary system of government, modelled on that of the United Kingdom. The people in each electoral district vote for a candidate of their choice. The candidate who receives the most votes becomes Members of Parliament (MP) for that electoral district. After an election, the party with the most elected representatives becomes the party in power.

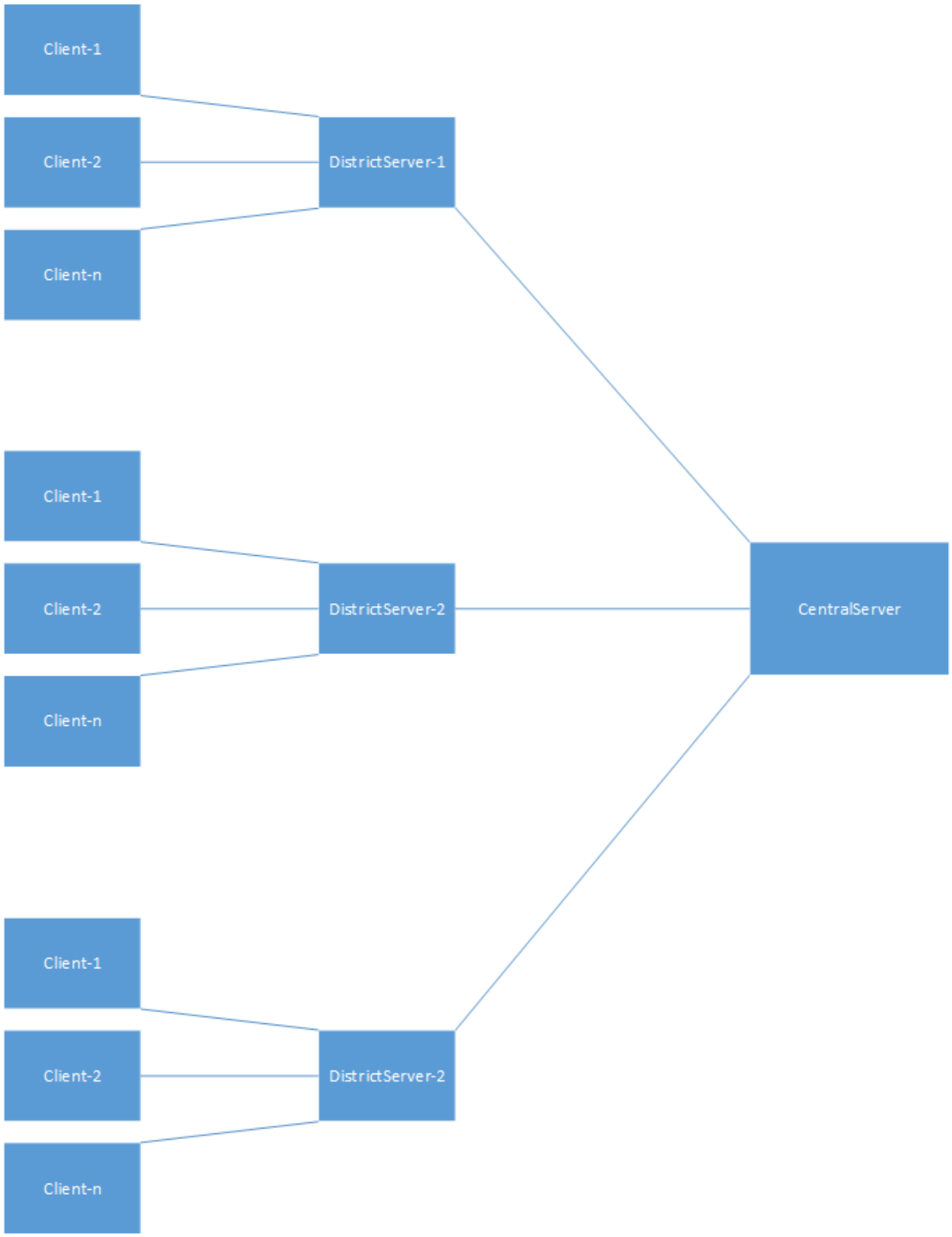
1 System Architecture

The Real-Time Electronic Voting System is programmed in Java as a distributed Client-Server system. A number of Clients connect to servers which are representative of their local electoral district. The Client systems are of a MVC architecture, and act as a polling station for voters to register and vote once for their chosen candidate and party. The client software allows voters to see the current statistics of the election in real-time, and register and vote by validating with a local District Server.

The District Servers processes all communication with the Clients in their district. They store and validate the Client's registration, votes, and compute statistics on those votes. A voter may only register through the Client once, and they need to login with the server in order to vote. Once they have voted, the District Server will not allow them to vote again.

The DistrictServers connect to a single CentralServer, which updates the entire federal election results in soft real-time. The Central Server queries all of the District Servers periodically for their current electoral statistics, and uses these statistics to compute the results of the federal election. That is, it determines which party has the most votes nation-wide. The system's distributed Client-Server architecture is depicted in Figure 1.1 below.

Figure 1.1: The system architecture diagram

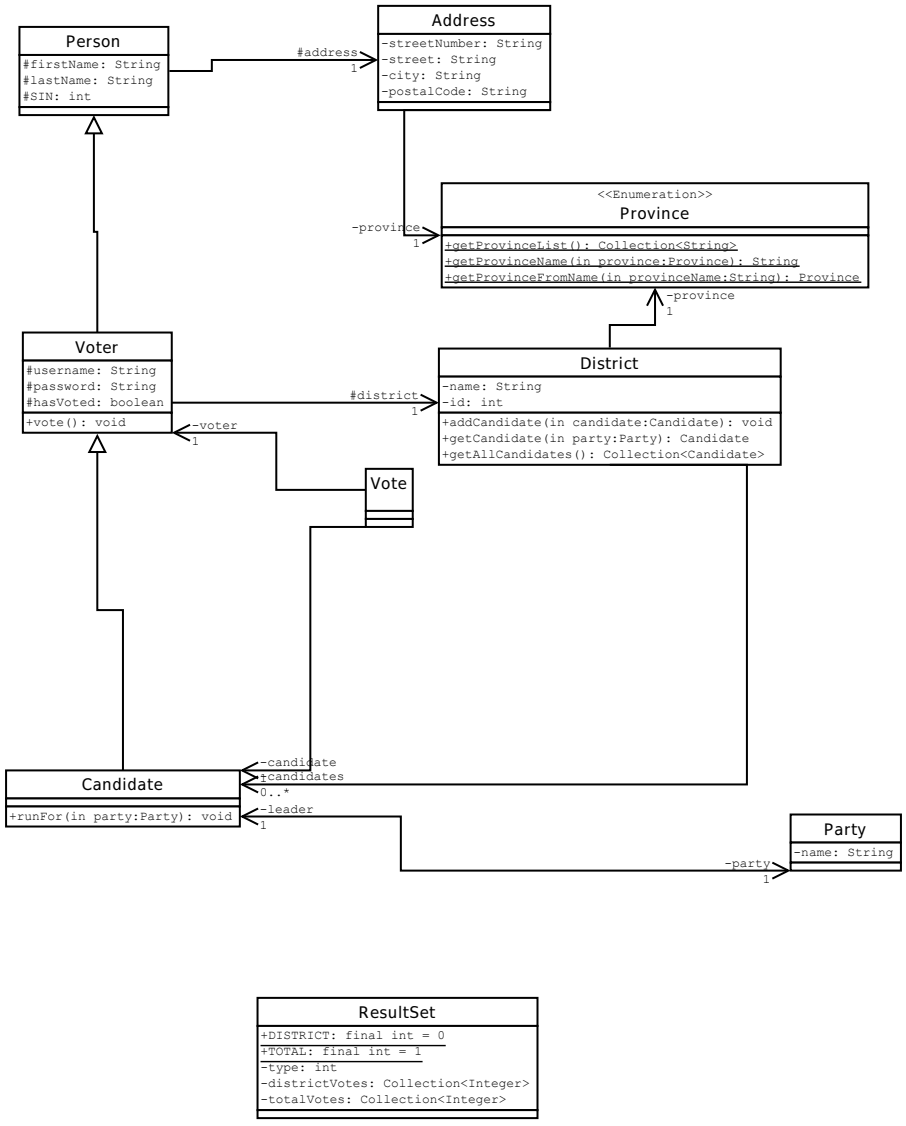


2 Java Class Implementation

The System is implemented over a collection of Java classes, which are organized into Java packages. The packages are `controller`, `model`, `networking`, and `view`. The package organization corresponds to a typical Model-View-Controller design. The classes contained within each package, and their relationships, are illustrated in the UML class diagram found at Figure 2.1.

Figure 2.1: UML class digarams

MODEL:

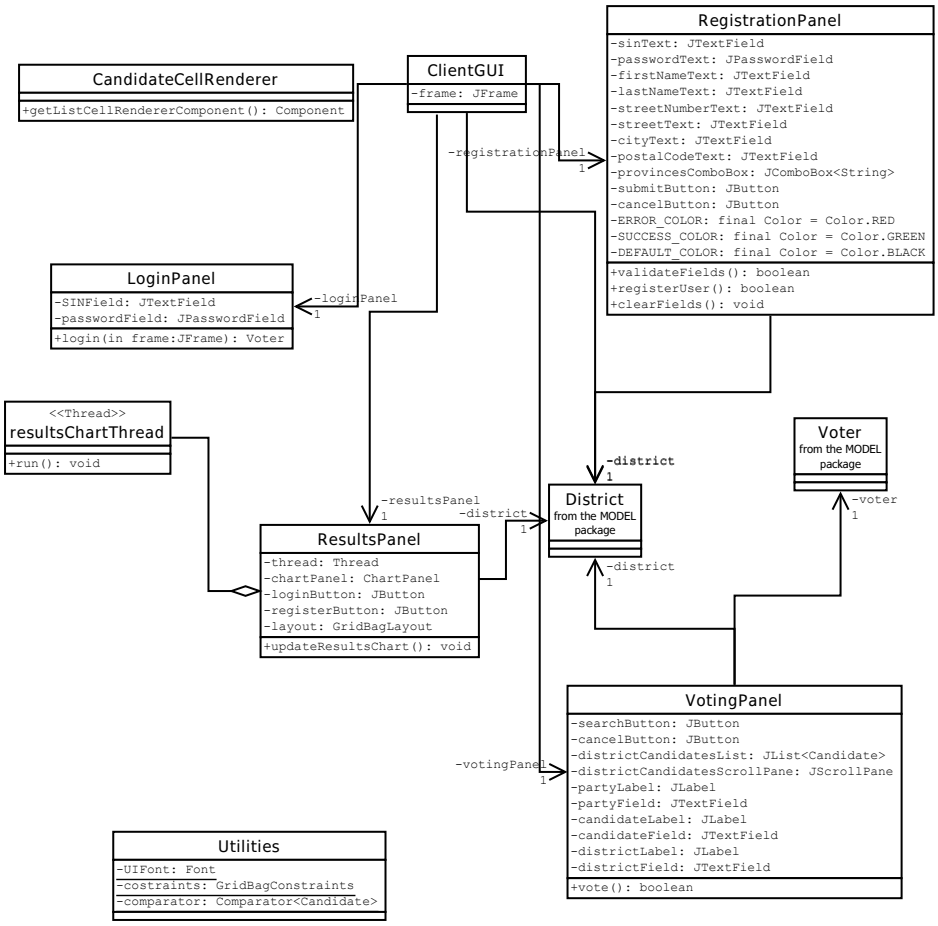

Person
#firstName: String
#lastName: String
#SIN: int
#address
1
Address
-streetNumber: String
-street: String
-city: String
-postalCode: String
<<Enumeration>>
Province
+getProvinceList(): Collection<String>
+getProvinceName(in province:Province): String
+getProvinceFromName(in provinceName:String): Province
-province
1
-province
1
Voter
#username: String
#password: String
#hasVoted: boolean
+vote(): void
#district
1
-voter
1
District
-name: String
-id: int
+addCandidate(in candidate:Candidate): void
+getCandidate(in party:Party): Candidate
+getAllCandidates(): Collection<Candidate>
Vote
-candidate
+candidates
0..*
Candidate
+runFor(in party:Party): void
-leader
1
Party
-name: String
-party
1

ResultSet
+DISTRICT: final int = 0
+TOTAL: final int = 1
-type: int
-districtVotes: Collection<Integer>
-totalVotes: Collection<Integer>

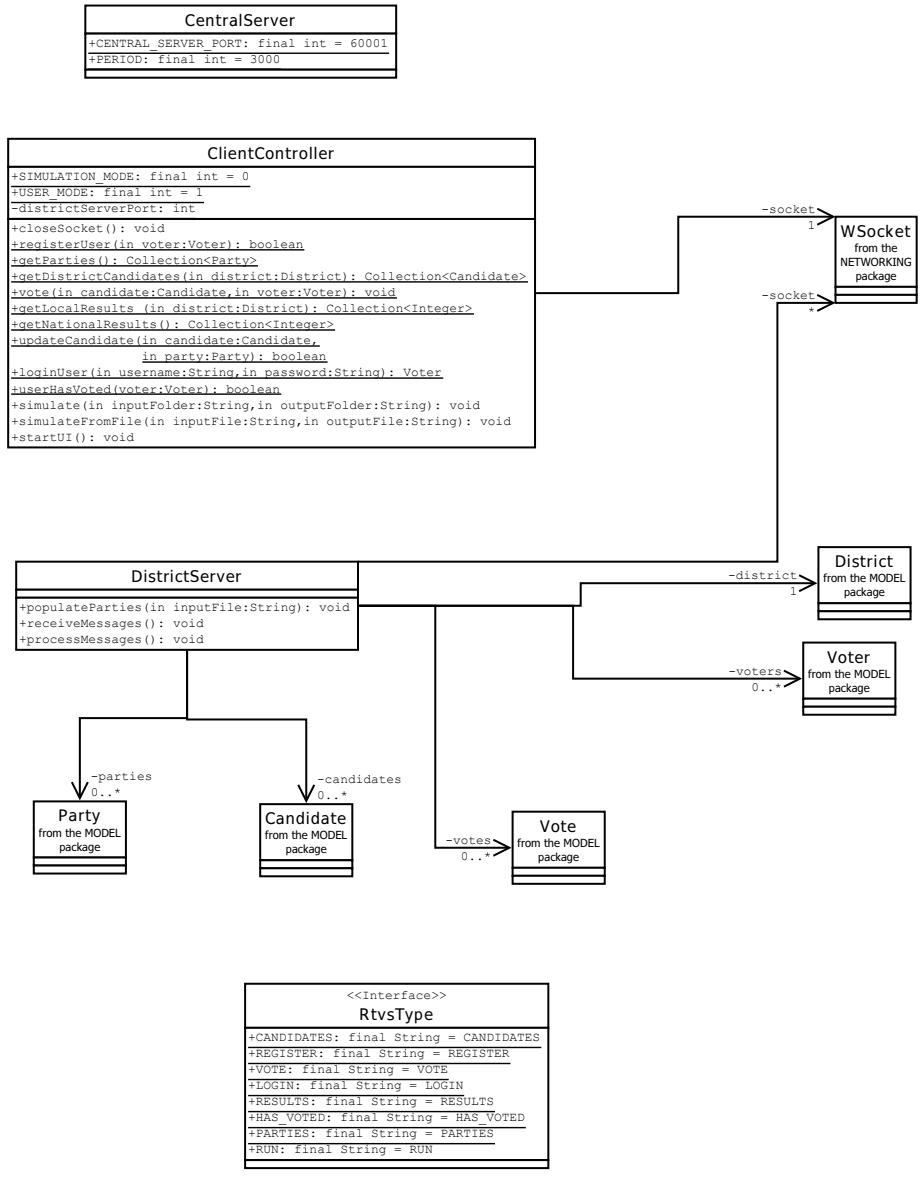
NETWORK:

Message
+Method: enum = {GET, POST, PUT, DELETE, CONNECT}
-data: Object
-method: Method
-checksum: long
-length: int
-senderPort: int
-type: Type
-senderAddr: InetAddress
+calculateChecksum(in data:Object): long
+getBytes(data:Object): byte[]
+getBytes(): byte[]
-calculateLength(): void
<<Exception>>
MessageCorruptException
WSocket
+PACKET_LEN: final int = 500
+FRAG_LEN: final int = 200
-TIMEOUT: int = 10
-socket: DatagramSocket
-addr: InetAddress
-port: int
+listen(in port:int,in host:String): WSocket
+listen(in port:int): WSocket
+connect(in port:int,in host:String): WSocket
+connect(,in port:int): WSocket
+close(): void
+receive(): Message
-sendConfirmation(in port:int,in host:InetAddress): void
+send(in msg:Message): void
+sendTo(in msg:Message,in port:int): void
+sendReceive(in msg:Message,in port:int,
in host:InetAddress): Message
+sendReceive(in msg:Message): Message
-receiveFragments(): Message
-sendFragments(in msg:Message,in port:int,
in host:InetAddress): void

VIEW:


CandidateCellRenderer
+getListCellRendererComponent(): Component
ClientGUI
-frame: JFrame
RegistrationPanel
-sinText: JTextField
-passwordText: JPasswordField
-firstNameText: JTextField
-lastNameText: JTextField
-streetNumberText: JTextField
-streetText: JTextField
-cityText: JTextField
-postalCodeText: JTextField
-provincesComboBox: JComboBox<String>
-submitButton: JButton
-cancelButton: JButton
-ERROR_COLOR: final Color = Color.RED
-SUCCESS_COLOR: final Color = Color.GREEN
-DEFAULT_COLOR: final Color = Color.BLACK
+validateFields(): boolean
+registerUser(): boolean
+clearFields(): void
-registrationPanel
1
LoginPanel
-SINField: JTextField
-passwordField: JPasswordField
+login(in frame:JFrame): Voter
-loginPanel
1
<<Thread>>
resultsChartThread
+run(): void
Voter
from the MODEL package
-district
1
-voter
1
-resultsPanel
1
-district
1
District
from the MODEL package
ResultsPanel
-thread: Thread
-chartPanel: ChartPanel
-loginButton: JButton
-registerButton: JButton
-layout: GridBagLayout
+updateResultsChart(): void
-district
1
VotingPanel
-searchButton: JButton
-cancelButton: JButton
-districtCandidatesList: JList<Candidate>
-districtCandidatesScrollPane: JScrollPane
-partyLabel: JLabel
-partyField: JTextField
-candidateLabel: JLabel
-candidateField: JTextField
-districtLabel: JLabel
-districtField: JTextField
+vote(): boolean
-votingPanel
1
Utilities
-UIFont: Font
-costraints: GridBagConstraints
-comparator: Comparator<Candidate>

CONTROLLER:


CentralServer
+CENTRAL_SERVER_PORT: final int = 60001
+PERIOD: final int = 3000
ClientController
+SIMULATION_MODE: final int = 0
+USER_MODE: final int = 1
-districtServerPort: int
+closeSocket(): void
+registerUser(in voter:Voter): boolean
+getParties(): Collection<Party>
+getDistrictCandidates(in district:District): Collection<Candidate>
+vote(in candidate:Candidate,in voter:Voter): void
+getLocalResults (in district:District): Collection<Integer>
+getNationalResults(): Collection<Integer>
+updateCandidate(in candidate:Candidate,
in party:Party): boolean
+loginUser(in username:String,in password:String): Voter
+userHasVoted(voter:Voter): boolean
+simulate(in inputFolder:String,in outputFolder:String): void
+simulateFromFile(in inputFile:String,in outputFile:String): void
+startUI(): void
-socket
1
WSocket
from the NETWORKING package
-socket
*
DistrictServer
+populateParties(in inputFile:String): void
+receiveMessages(): void
+processMessages(): void
-district
1
District
from the MODEL package
-voters
0..*
Voter
from the MODEL package
-parties
0..*
Party
from the MODEL package
-candidates
0..*
Candidate
from the MODEL package
-votes
0..*
Vote
from the MODEL package
<<Interface>>
RtvsType
+CANDIDATES: final String = CANDIDATES
+REGISTER: final String = REGISTER
+VOTE: final String = VOTE
+LOGIN: final String = LOGIN
+RESULTS: final String = RESULTS
+HAS_VOTED: final String = HAS_VOTED
+PARTIES: final String = PARTIES
+RUN: final String = RUN

3 The Communications Protocol

The system's communication protocol is handled by a networking subsystem that simulates the TCP protocol over a UDP connection. This networking component contains a java class named, `WSocket` (which stands for wrapper-socket). This class offers a `send` instance method which guarantees that message being passed are received by another instance of `WSocket`. It achieves this by sending a packet over a DatagramSocket, then waiting for another packet in response to signify that the data was received and intact. If no response is received, a `TimeoutException` is thrown and the message is resent. The `WSocket` class also offers a `receive` method, which blocks and waits for a message to be received over a datagram socket. When the message is received, it is deserialized and a CRC checksum is calculated on the contained data. If the checksum indicates that the data is corrupted, it does not send any notification message back to the sender. This effectively causes the sender to resend the same message, since they will timeout after not receiving a response. If the data is intact, a notification message is sent back to the sender and the deserialized message is returned to the receiver.

There is another limitation of using a UDP connection; and that is that individual packets can only transport a small amount of data. The `WSocket` class therefore breaks messages that exceed the size of one packet into fragments. Fragments are sent and received in order, following the same protocol outlined above, then are pieced back together by the receiver to reconstruct the initial message. This protocol is illustrated in the two sequence diagrams below. Figure 3.1 shows how the protocol operates for the sender, which may be a Client in the Real-Time Electronic Voting System, and Figure 3.2 shows the protocol as it works for the receiver, which may be a server in the system. Note that the exception handling, in particular, for the timeout and message-corrupted exceptions, are not shown in these diagrams.

Figure 3.1: Sequence Diagram showing a sender's general usage of the socket

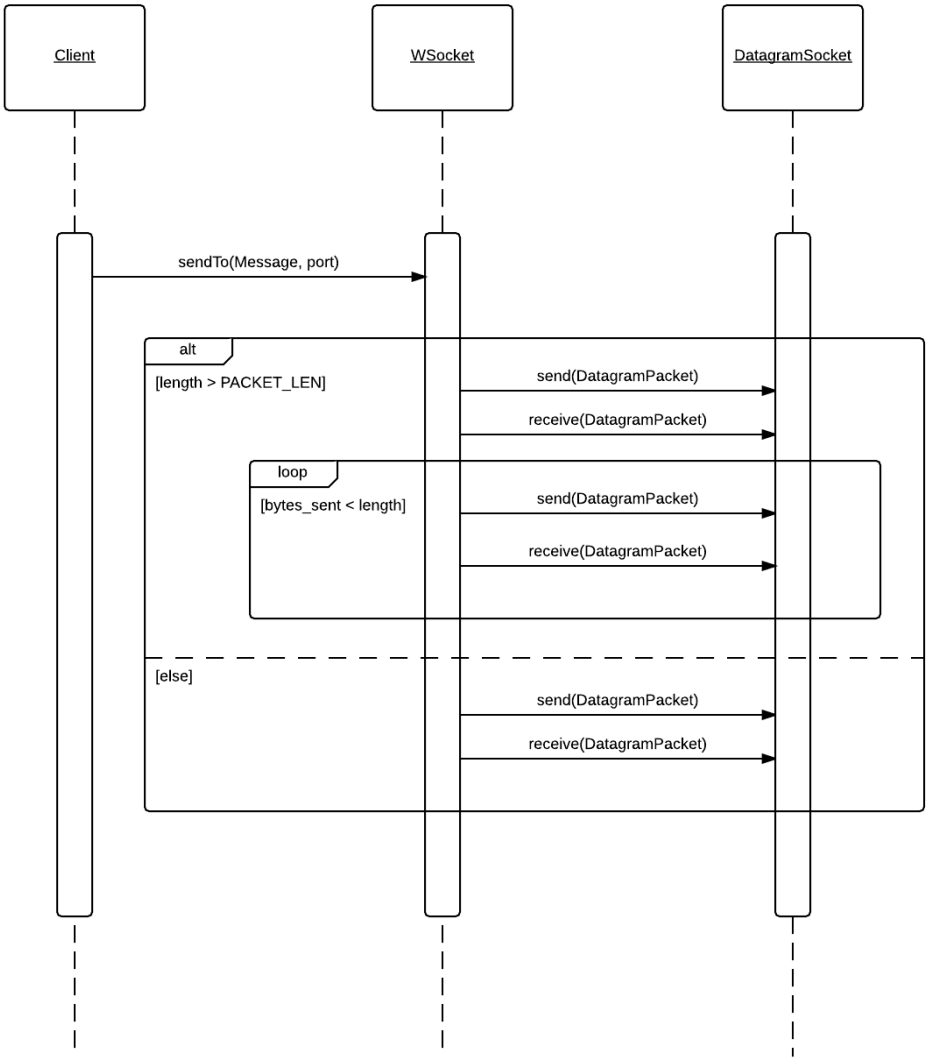
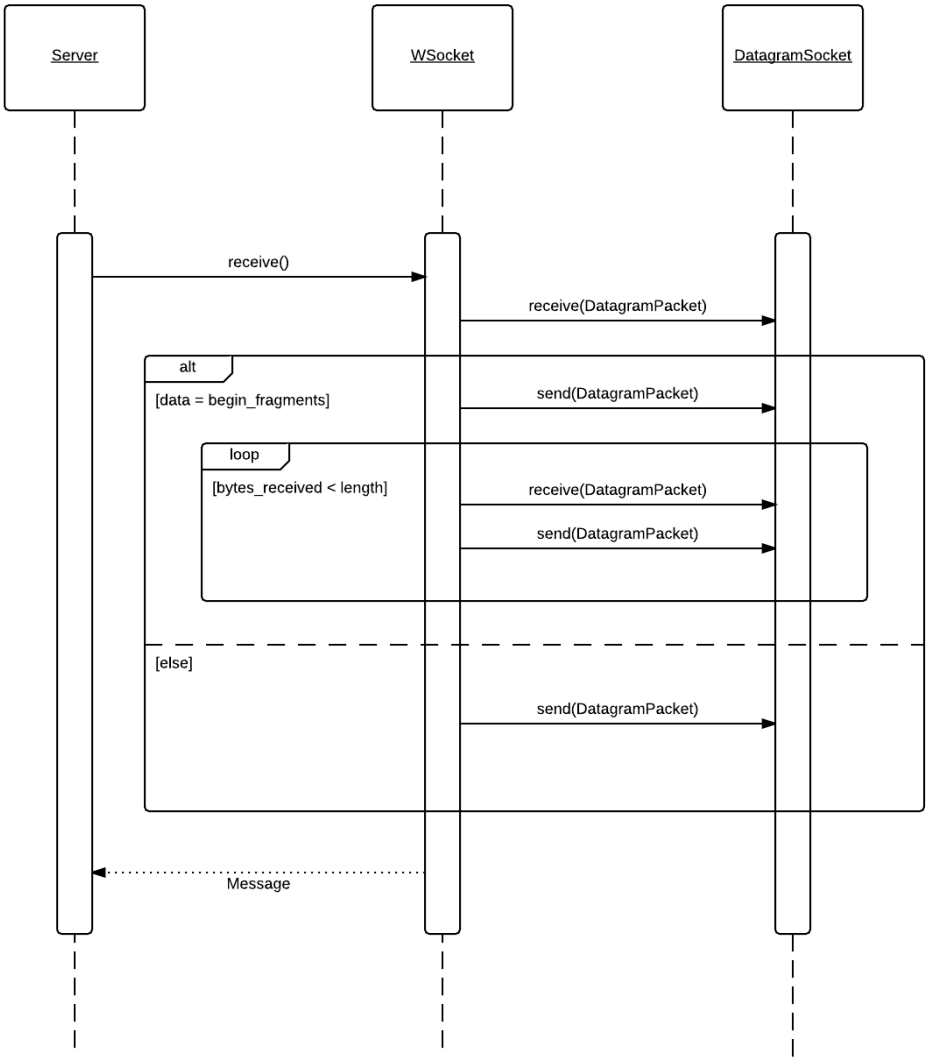


Figure 3.2: Sequence Diagram showing a receiver's general usage of the socket



More specifically, the Real-Time Electronic Voting System uses the networking subsystem to send a finite set of message types. A typical interaction between a Client and District Server may involve sending an odered sequence of messages. For example, a Client will first request the list candidates running in that district. A voter may then register with the District Server, then login, before they can send their vote. Meanwhile, a background thread started by the Client periodically pings the current election results from the server. This typical interaction is shown in the sequenec diagram below.

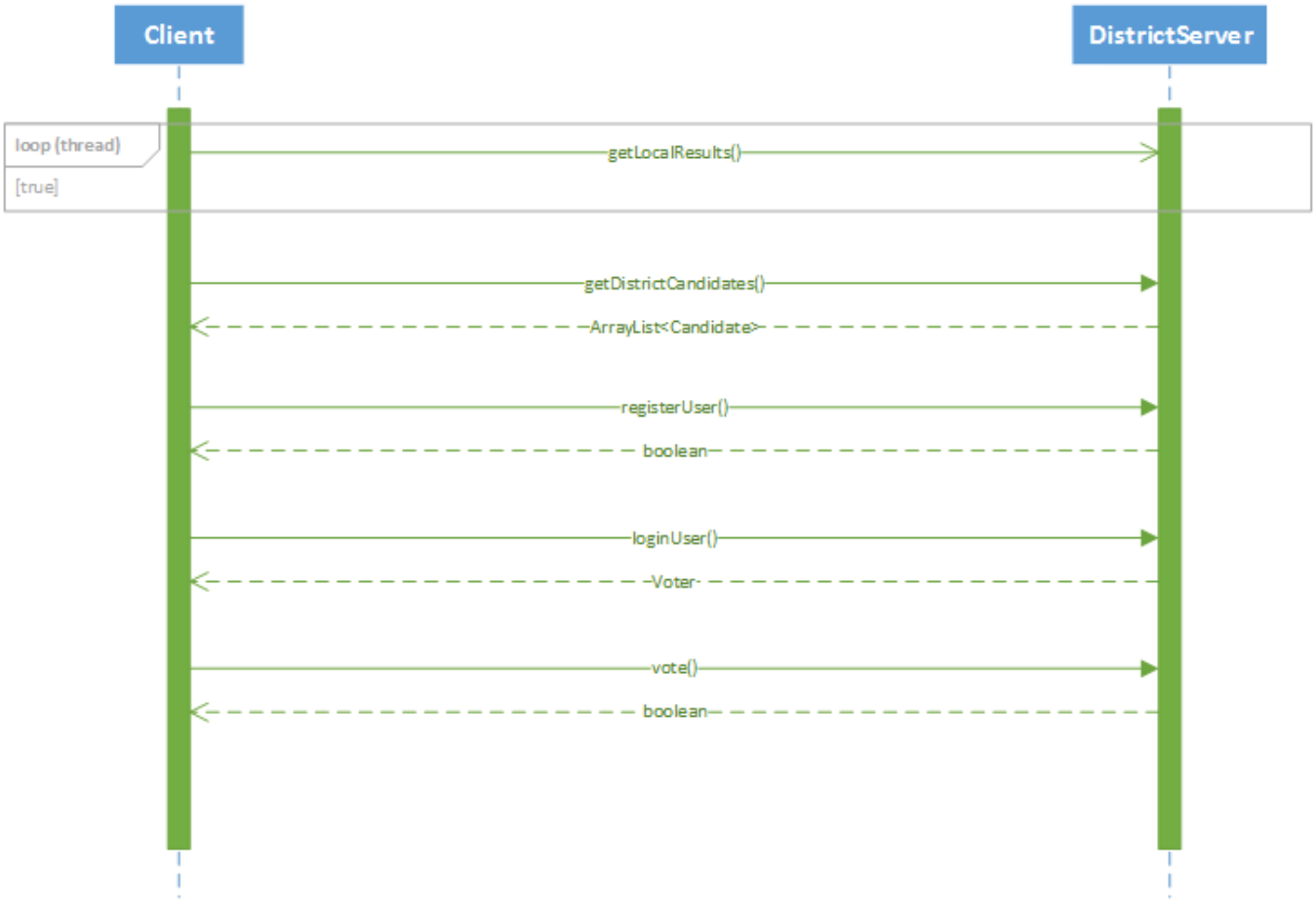



Figure 3.3: Typical message passing between a Client and the District Server