ΚΕΦΑΛΑΙΟ II

Εισαγωγή στον προγραμματισμό κατανεμημένων συστημάτων λογισμικού

Σύνοψη

Το κεφάλαιο εισάγει τον αναγνώστη στον προγραμματισμό κατανεμημένων συστημάτων λογισμικού. Στο κεφάλαιο αυτό πραγματευόμαστε το μοντέλο πελάτη-διακομιστή (client-server), το προγραμματιστικό πλαίσιο Twisted το οποίο διευκολύνει σημαντικά τον προγραμματισμό κατανεμημένων συστημάτων, και ένα παράδειγμα προγραμματισμού της πλευράς του πελάτη στο κατανεμημένο σύστημα αποθήκευσης Cassandra.

Προαπαιτούμενη γνώση

Η κατανόηση των εννοιών και παραδειγμάτων σε αυτό το κεφάλαιο απαιτεί βασική γνώση της γλώσσας Python και του δικτυακού προγραμματισμού, όπως αντίστοιχα παρουσιάστηκαν στα Κεφάλαια I-8 και I0 του παρόντος συγγράμματος.

II.I Εισαγωγή

Ένα *κατανεμημένο σύστημα* ορίζεται ως ένα σύστημα στο οποίο μέρη λογισμικού που βρίσκονται σε διαφορετικούς, δικτυωμένους υπολογιστές, επικοινωνούν και συντονίζουν τις δράσεις τους μέσω ανταλλαγής μηνυμάτων (Coulouris, Dollimore, & Kindberg, 2005). Στο Κεφάλαιο I0 μελετήσαμε τρόπους επικοινωνίας πάνω από δίκτυο μέσω της διεπαφής των sockets. Ο προγραμματισμός κατανεμημένων συστημάτων με κατευθείαν χρήση sockets, ωστόσο, οδηγεί σε προγράμματα υψηλής

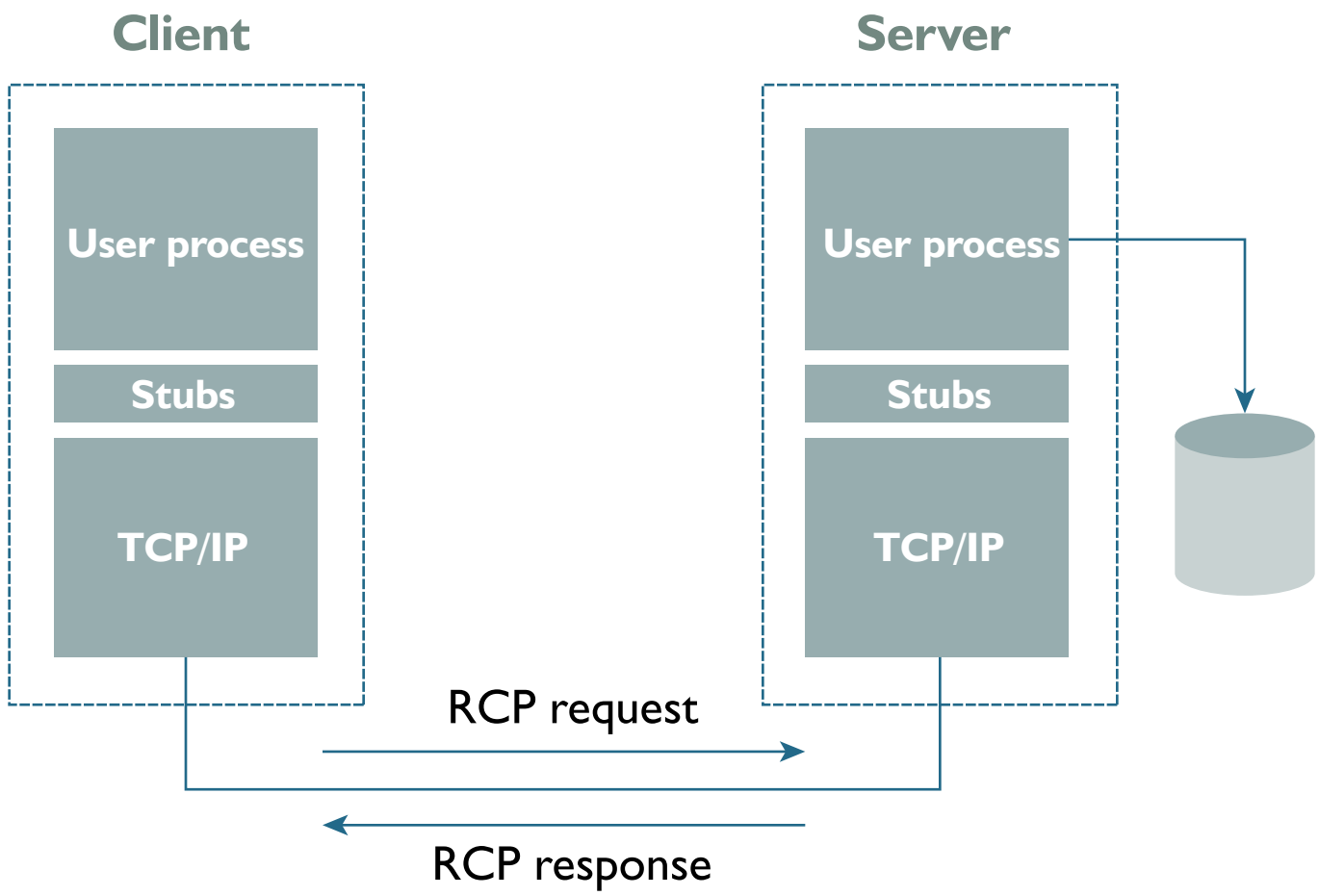
πολυπλοκότητας (όπως της Εικόνας 10.10), και άρα χρειαζόμαστε έναν ευκολότερο τρόπο.

Ένα διαδεδομένο μοντέλο κατανεμημένου συστήματος είναι αυτό του πελάτη-διακομιστή (client-server). Ο προγραμματισμός αυτού του μοντέλου κατανεμημένου συστήματος γίνεται συχνά με μια διεπαφή εφαρμογής-χρήστη υψηλότερου επιπέδου από τα sockets, αυτή των *απομακρυσμένων κλήσεων διαδικασιών* (remote procedure calls ή RPCs) (Birrell & Nelson, 1983). Ένας σχεδιαστικός στόχος των RPCs είναι να δώσουν την αίσθηση στον προγραμματιστή ότι εκτελεί μια *τοπική κλήση συνάρτησης*, αφαιρώντας την ανάγκη να χειριστεί ο ίδιος την πολυπλοκότητα της αποστολής και λήψης μηνυμάτων. Αυτό το χειρίζεται εσωτερικά η υλοποίηση των RPCs, αποστέλλοντας μηνύματα αίτησης (request) και απάντησης (response) πάνω από (συνήθως) TCP/IP συνδέσεις (Εικόνα 11.1). Οι υλοποιήσεις RPC παράγουν, αυτόματα, κώδικα ο οποίος χειρίζεται την εισαγωγή/εξαγωγή παραμέτρων και αποτελεσμάτων από τα μηνύματα αυτά, ώστε να μη χρειαστεί να το κάνει αυτό ο προγραμματιστής. Ο κώδικας αυτός συχνά αναφέρεται και ως stubs (Εικόνα 11.1).

Ο αρχικός σκοπός, λοιπόν, των RPCs ήταν να μπορεί να εστιάσει ο προγραμματιστής, κυρίως, στη λογική της εφαρμογής του, σαν να προγραμμάτιζε μία μη-κατανεμημένη εφαρμογή. Η φύση, όμως, ενός κατανεμημένου συστήματος καθιστά δύσκολο αυτό το έργο: Εφόσον οι υπολογιστές που τρέχουν τα client και server μέρη του συστήματος, μπορεί να καταρρεύσουν ανεξάρτητα (δηλαδή, να καταρρεύσει ο server και όχι ο client ή αντίστροφα), είναι δυνατόν η εκτέλεση ενός RPC να τερματίσει με μια *εξαίρεση* που μπορεί να οφείλεται σε αστοχία στο δίκτυο ή στο διακομιστή πριν ή μετά την εκτέλεση της απομακρυσμένης διαδικασίας.

Στην περίπτωση κατά την οποία ο πελάτης λάβει μια εξαίρεση κατά την εκτέλεση του RPC, δεν μπορεί να είναι σίγουρος αν η αστοχία έχει συμβεί *πριν* ή *μετά* την εκτέλεση της απομακρυσμένης διαδικασίας. Οι επιλογές του πελάτη είναι είτε να υποθέσει ότι δεν εκτελέστηκε και άρα να επαναλάβει την αίτηση μέχρι να λάβει επιτυχή απάντηση ή να θεωρήσει ότι εκτελέστηκε και άρα να μην επαναλάβει την αίτηση. Στην πρώτη περίπτωση λέμε ότι η υλοποίηση RPC εκτελεί την απομακρυσμένη διαδικασία «τουλάχιστον μια φορά» (τη φορά που έλαβε την επιτυχή απάντηση αλλά πιθανώς και σε κάποιες άλλες από τις προσπάθειες που έλαβε εξαί-

ρεση) ενώ στη δεύτερη περίπτωση «το μέγιστο μια φορά» (αν η απομακρυσμένη συνάρτηση εκτελέστηκε στη μία αυτή προσπάθεια).



Εικόνα II.1 *Λειτουργία απομακρυσμένης κλήσης διαδικασίας (RPC) στο μοντέλο πελάτη-εξυπηρετητή*

Υλοποιήσεις των RPCs έχουν χρησιμοποιηθεί σε πολλά και σημαντικά συστήματα, όπως το Network File System (NFS), και χρησιμοποιείται σήμερα για την υλοποίηση πολλών κατανεμημένων συστημάτων, όπως το σύστημα Cassandra που πραγματευόμαστε στην Ενότητα II.3.

Παρά τα σημαντικά οφέλη του μοντέλου των RPCs, εφαρμογές που απαιτούν κυρίως τη μεταφορά δεδομένων και όχι την κλήση απομακρυσμένων διαδικασιών, ιδιαίτερα αυτές που εμπλέκουν επικοινωνία εντός μιας ομάδας διεργασιών (όπως η πολυεκπομπή που είδαμε στο Κεφάλαιο I0), ταιριάζουν περισσότερο σε ένα μοντέλο ανταλλαγής μηνυμάτων. Μια τέτοια διεπαφή η οποία μπορεί να χρησιμοποιηθεί για την ανταλλαγή μηνυμάτων, με χαμηλότερη πολυπλοκότητα κώδικα από αυτήν των sockets, προσφέρεται από το *πλαίσιο προγραμματισμού (programming framework)* Twisted, το οποίο θα δούμε στην Ενότητα II.2.

Η έννοια του *προγραμματιστικού πλαισίου* γενικότερα στοχεύει στην απλοποίηση του προγραμματισμού συστημάτων και της γρήγορης παραγωγής πρότυπου κώδι-

κα (Wayner, 2015). Ένα προγραμματιστικό πλαίσιο παρέχει υποστηρικτικό κώδικα ο οποίος μειώνει αρκετά τον κώδικα που πρέπει να παρέχει ο χρήστης, επιτρέποντας στον τελευταίο να εστιάσει στη λογική της εφαρμογής. Το προγραμματιστικό πλαίσιο, συνήθως, διαχειρίζεται τον έλεγχο ροής του προγράμματος και παρέχει μια όσο το δυνατόν απλούστερη διεπαφή χρήστη-πλαισίου.

Ο τρόπος επικοινωνίας με χρήση του πλαισίου Twisted είναι υψηλότερου επιπέδου από τα sockets. Μοιάζει με το μοντέλο RPC στο ότι η κλήση μιας συνάρτησης αποστολής μηνύματος στην πλευρά του πελάτη, προκαλεί *αυτόματα* την κλήση μιας συνάρτησης λήψης μηνύματος στην πλευρά του διακομιστή (χωρίς ο τελευταίος να χρειαστεί να καλέσει socket.recv). Διαφέρει, ωστόσο, από τα RPCs στο ότι η ανταλλαγή μηνυμάτων υλοποιείται από την εφαρμογή. Με χρήση του πλαισίου Twisted μπορεί να υλοποιηθούν σημαντικές κατανεμημένες εφαρμογές, όπως διάφορες μορφές πολυεκπομπής (Coulouris et al., 2005).

II.2 Το πλαίσιο δικτυακού προγραμματισμού Twisted

Το πλαίσιο προγραμματισμού Twisted υποστηρίζει το δικτυακό προγραμματισμό, με βάση τη διαχείριση γεγονότων και την ασύγχρονη κλήση μεθόδων επικοινωνίας. Κεντρικές έννοιες στο Twisted είναι ο *βρόγχος γεγονότων* (event loop), εντός του οποίου εκτελεί, διαρκώς, το μοναδικό νήμα εκτέλεσης, και η βασική κλάση Protocol, την οποία επεκτείνει κάθε πρωτόκολλο επιπέδου εφαρμογής (όπως το Peer το οποίο θα αναλύσουμε παρακάτω).

Στην Εικόνα II.2 βλέπουμε μια γραφική απεικόνιση των βασικών χαρακτηριστικών της χρήσης του Twisted. Η απεικόνιση αυτή εστιάζει σε χαρακτηριστικά που θα χρησιμοποιήσουμε στη συνέχεια αυτού του κεφαλαίου και όχι σε μια πλήρη παρουσίαση του Twisted (ο ενδιαφερόμενος αναγνώστης μπορεί να συμβουλευτεί τον σύνδεσμο (Twisted Matrix Labs, 2015)). Ο χαρακτηρισμός ενός μέρους ως client ή server έχει να κάνει με το ποιος ξεκίνησε πρώτος το στήσιμο της σύνδεσης καλώντας connect και ποιος αποδέχθηκε την αίτηση σύνδεσης αντίστοιχα. Μετά το στήσιμο της σύνδεσης, οι δύο πλευρές εκτελούν το βρόγχο γεγονότων.

Ο βρόγχος αναφέρεται και ως reactor (το αντίστοιχο module ονομάζεται twisted.internet.reactor), λόγω της φύσης λειτουργίας του, η οποία είναι *να αντιδρά* (react) σε εισερχόμενα γεγονότα.

Ο βρόγχος γεγονότων λαμβάνει ως είσοδο γεγονότα όπως *εισερχόμενα δεδομένα, αιτήσεις για στήσιμο σύνδεσης*, ή *ειδοποιήσεις τερματισμού σύνδεσης*. Κάθε γεγονός αντιστοιχίζεται στη σύνδεση στην οποία αναφέρεται, η οποία με τη σειρά της αντιστοιχεί σε ένα αντικείμενο τύπου Protocol (ένα σε κάθε πλευρά της σύνδεσης), που με τη σειρά του συνδέεται με το αντίστοιχο αντικείμενο τύπου Protocol στην επικοινωνούσα διεργασία. Το αντικείμενο δημιουργείται κατά το στήσιμο της σύνδεσης από την κλάση-εργοστάσιο ProtocolFactory. Κάθε δικτυακή εφαρμογή (όπως οι Echo και Peer που θα δούμε στη συνέχεια) δημιουργεί μια υποκλάση της Protocol, οι κύριες μέθοδοι της οποίας είναι οι παρακάτω χειριστές (handlers) των γεγονότων:

connectionMade(self): Καλείται, όταν η σύνδεση έχει στηθεί επιτυχώς.

dataReceived(self, data): Καλείται, όταν υπάρχουν εισερχόμενα δεδομένα.

connectionLost(self, reason): Καλείται, όταν διακοπεί η σύνδεση.

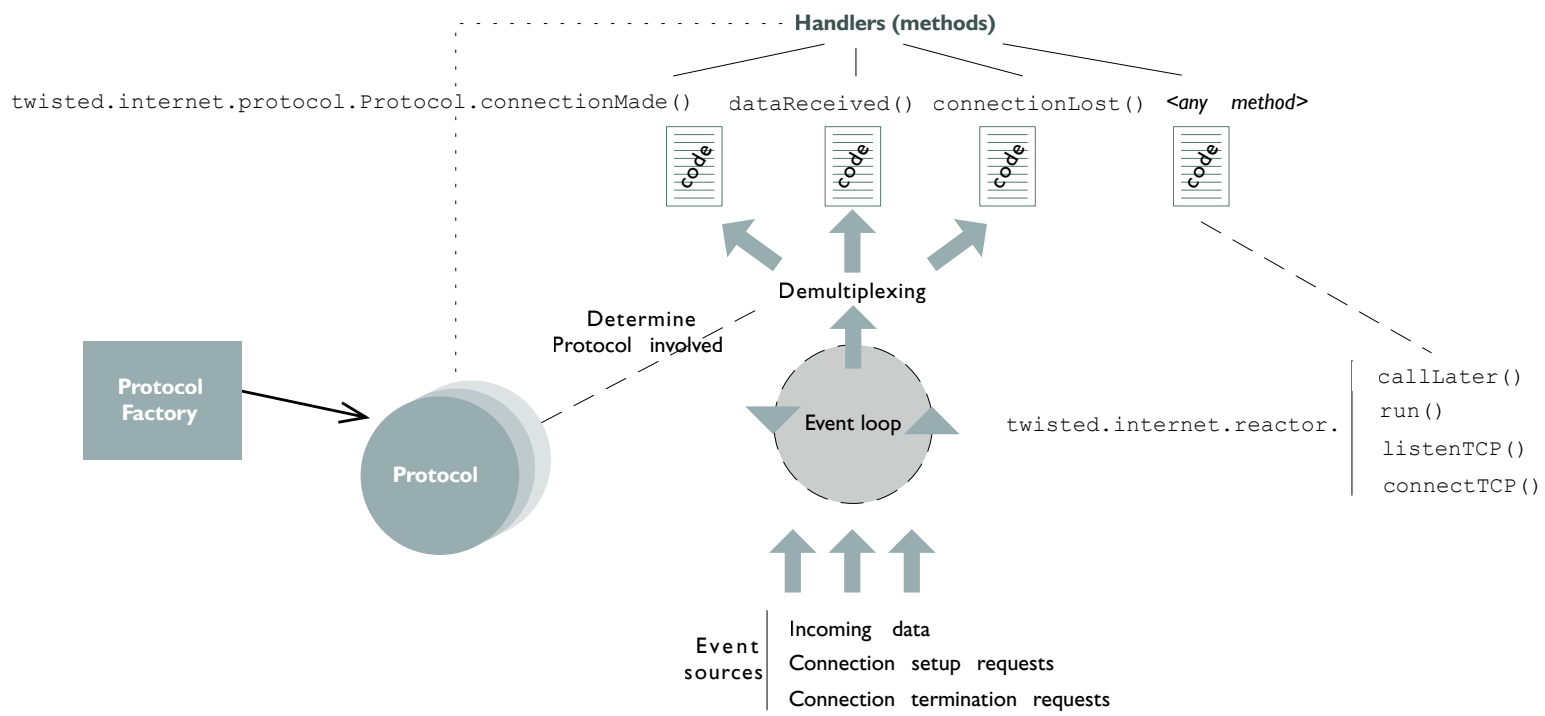
Επιπρόσθετα, κάθε υποκλάση της Protocol ορίζει τη μέθοδο __init__(self, factory, p) η οποία καλείται από την ProtocolFactory κατά τη δημιουργία του αντικειμένου.

Εκτός από τις μεθόδους-χειριστές (handlers), άλλες χρήσιμες μέθοδοι που προσφέρει ο βρόγχος γεγονότων (reactor) του Twisted, και οι οποίες καλούνται απευθείας από τον κώδικα είναι:

reactor.callLater(sec, method): Καλεί τη μέθοδο method (ορισμένη από το χρήστη) εντός συγκεκριμένου χρονικού διαστήματος (sec). Αυτή η συνάρτηση είναι χρήσιμη για την περιοδική εκτέλεση λειτουργιών, όπως η περιοδική αποστολή μηνυμάτων τύπου “heartbeat”.

reactor.listenTCP(port, factory): Κατ’ αντιστοιχία με το socket.listen που είδαμε στο Κεφάλαιο I0, η listenTCP ζητά από το βρόγχο να ακούει στην πόρτα port. Όταν ολοκληρωθεί κάποια σύνδεση, δημιουργείται νέο αντικείμενο τύπου Protocol (από την κλάση ProtocolFactory (όρισμα factory)).

reactor.connectTCP(host, port, factory): Κατ' αντιστοιχία με το socket.connect που είδαμε στο Κεφάλαιο 10, η connectTCP ζητά από το βρόγχο να ξεκινήσει μια σύνδεση προς τη διεύθυνση host και πόρτα port. Όταν ολοκληρωθεί η σύνδεση, δημιουργείται νέο αντικείμενο τύπου Protocol (από την κλάση ProtocolFactory (όρισμα factory)).



Εικόνα 11.2 *Λειτουργία του βρόγχου γεγονότων του Twisted*

Στην Εικόνα 11.3 θα δούμε μια πιο εκτεταμένη γραφική απεικόνιση των βασικών χαρακτηριστικών και της τυπικής λειτουργίας του client μέρους με πολλαπλές συνδέσεις.

Η αποστολή δεδομένων πάνω από μια σύνδεση την οποία ενθυλακώνει αντικείμενο τύπου Protocol γίνεται μέσω της συνάρτησης transport.write(data) η οποία προσφέρεται από τη γονική κλάση Protocol.

Αν ο προγραμματιστής θέλει να εξειδικεύσει τη δημιουργία αντικειμένων Protocol, τότε πρέπει να δημιουργήσει μια υποκλάση της ProtocolFactory η οποία να κάνει ακριβώς αυτό. Οι μέθοδοι που, συνήθως, χρειάζεται να υλοποιηθούν σε αυτήν την περίπτωση είναι οι:

- buildProtocol(self, addr): Καλείται, για να δημιουργήσει ένα εξειδικευμένο αντικείμενο τύπου Protocol.

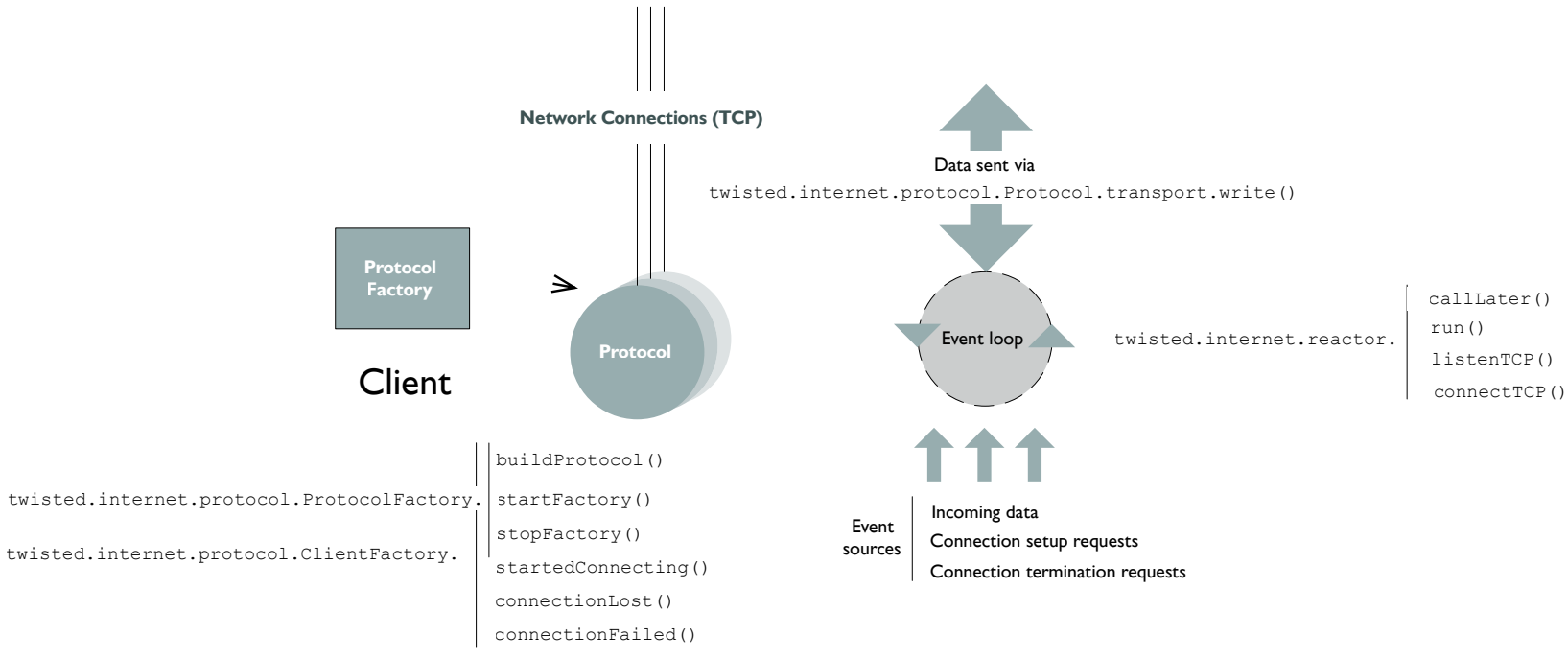
- startFactory(self): Καλείται, πριν ζητηθεί από το βρόγχο να ακούει σε κάποια πόρτα. Αυτή η μέθοδος καλείται μόνο μία φορά για να εκτελέσει αρχικές δράσεις, ακόμα και αν ο βρόγχος ακούει σε περισσότερες της μίας πόρτες.
- stopFactory(self): Καλείται, πριν ζητηθεί από το βρόγχο να σταματήσει να ακούει σε κάποια πόρτα. Αυτή η μέθοδος καλείται, συνήθως, για να απελευθερώσει πόρους.

Μια υποκλάση της ProtocolFactory, την οποία το Twisted προσφέρει έτοιμη, είναι η ClientFactory η οποία παράγει αντικείμενα τύπου ClientProtocol, εξειδικευμένα για λειτουργίες τύπου client. Μέθοδοι της ClientFactory (επιπρόσθετες των παραπάνω της ProtocolFactory) είναι οι:

- startedConnecting(self, connector): Καλείται, όταν η διαδικασία του στησίματος μιας σύνδεσης έχει ξεκινήσει. Το πλαίσιο Twisted δεν ενθαρρύνει τη χρήση του αντικειμένου connector από τους προγραμματιστές εφαρμογών και έτσι δε θα ασχοληθούμε περαιτέρω με αυτό.
- clientConnectionFailed(self, connector, reason): Καλείται, όταν μια προσπάθεια για σύνδεση ήταν ανεπιτυχής.
- clientConnectionLost(self, connector, reason): Καλείται, όταν κάποια υπάρχουσα σύνδεση διακοπεί.

Η ClientFactory συνεργάζεται με τη μέθοδο reactor.connectTCP(host,port,factory) του βρόγχου γεγονότων. Μόλις μια σύνδεση έχει στηθεί μέσω της connectTCP, καλείται η μέθοδος startedConnecting.

Στη συνέχεια θα δούμε δύο παραδείγματα προγραμματισμού με χρήση του Twisted. Θα ξεκινήσουμε με ένα απλό παράδειγμα (κλάση Echo) και θα συνεχίσουμε με ένα λίγο πιο σύνθετο (κλάση Peer).



Εικόνα II.3 *Λειτουργία του client μέρους στο Twisted με πολλαπλές συνδέσεις*

II.2.1 Echo class

Η πρώτη μας εμπειρία με το πλαίσιο προγραμματισμού Twisted θα γίνει μέσω ενός απλού παραδείγματος, του γνωστού μας Echo (McKellar, 2013). Το παράδειγμα αυτό (Εικόνα II.4) υποδεικνύει τη χρήση των μεθόδων των Protocol και reactor για την υλοποίηση ενός διακομιστή echo[1]. Ο αναγνώστης καλείται να αντιπαραβάλει τη συνοπτικότητα και απλότητα του παραδείγματος αυτού με την υλοποίηση του ίδιου διακομιστή με χρήση του select (Εικόνα I0.II) ή των βασικών μεθόδων του socket API (Εικόνα I0.I).

[1] Ο κώδικας αυτού του κεφαλαίου έχει δοκιμαστεί με την έκδοση 14.0.2 του Twisted.

```python
import twisted.internet.protocol
import twisted.internet.reactor

class EchoProtocol(twisted.internet.protocol.Protocol):
    def connectionMade(self):
        self.peer = self.transport.getPeer( )[1:]
        print "Connected from", self.peer
    def dataReceived(self, data):
        self.transport.write(data)
    def connectionLost(self, reason):
        print "Disconnected from", self.peer, reason.value

factory = twisted.internet.protocol.Factory( )
factory.protocol = EchoProtocol

twisted.internet.reactor.listenTCP(8881, factory)
twisted.internet.reactor.run( )
```

Εικόνα ΙΙ.4 *Echo server*

Το EchoProtocol υλοποιεί τις τρεις μεθόδους-χειριστές (handlers) που περιγράφηκαν νωρίτερα. Στο κυρίως σώμα του προγράμματος καλούμε τον reactor σε δύο περιπτώσεις. Κατά την πρώτη για να του ζητήσουμε να ακούει στην πόρτα 8881, περνώντας του το ProtocolFactory που θα χρησιμοποιήσουμε για τη δημιουργία αντικειμένων Protocol σε κάθε σύνδεση. Σε αυτήν την περίπτωση χρησιμοποιούμε την standard υλοποίηση του ProtocolFactory, η οποία θα παράγει αντικείμενα τύπου EchoProtocol (factory.protocol = EchoProtocol). Στη δεύτερη κλήση (twisted.internet.reactor.run()) ξεκινάμε την αέναη εκτέλεση του βρόγχου γεγονότων. Σε αυτό το παράδειγμα, εστιάσαμε στην υλοποίηση ενός απλού server. Ο αναγνώστης μπορεί να δοκιμάσει την υλοποίηση του server χρησιμοποιώντας το πρόγραμμα telnet ή ssh προς την πόρτα 8881 (McKellar, 2013). Στο επόμενο παράδειγμα θα δούμε μια πληρέστερη υλοποίηση των μερών του client και server.

II.2.2 Peer class

Σε αυτήν την ενότητα θα μελετήσουμε την κλάση Peer, η οποία μπορεί να αποτελέσει δομικό λίθο για την υλοποίηση γενικότερων λειτουργιών συστημάτων κατανεμημένου λογισμικού. Η κλάση Peer μαζί με τις υποστηρικτικές κλάσεις και συναρτήσεις μπορεί να συμπεριφερθεί είτε ως πελάτης (συνδεόμενη σε ένα διακομιστή και, περιοδικά, στέλνοντας ένα μήνυμα ενημέρωσης προς αυτόν) ή ως διακομιστής (δεχόμενος τα περιοδικά μηνύματα από συνδεδεμένους πελάτες και απαντώντας σε καθένα από αυτά με μια επιβεβαίωση).

Το παράδειγμα της Εικόνας II.5 υποδεικνύει το χειρισμό των παραμέτρων εισόδου, οι οποίες είναι (α) ο τρόπος λειτουργίας της συγκεκριμένης εκτέλεσης (πελάτης (client) ή διακομιστής (server)), και (β) η IP διεύθυνση του μηχανήματος και πόρτα στην οποία θα συνδεθεί ο πελάτης. Χρησιμοποιώντας το module optparse, παίρνουμε τις παραμέτρους εισόδου σαν μια πλειάδα (args). Στη συνέχεια εξάγουμε τον τρόπο λειτουργίας (peer_type), και την IP διεύθυνση και πόρτα μέσω της συνάρτησης parse_address.

10-peer-twisted-server.py

```
11  import optparse
12
13  from twisted.internet.protocol import Protocol, ClientFactory
14  from twisted.internet import reactor
15  import time
16
17  def parse_args():
18      usage = """usage: %prog [options] [client|server] [hostname]:port
19
20      python peer.py server 127.0.0.1:port """
21
22      parser = optparse.OptionParser(usage)
23
24      _, args = parser.parse_args()
25
26      if len(args) != 2:
27          print parser.format_help()
28          parser.exit()
29
30      peertype, addresses = args
31
32      def parse_address(addr):
33          if ':' not in addr:
34              host = '127.0.0.1'
35              port = addr
36              else:
37              host, port = addr.split(':', 1)
38
39          if not port.isdigit():
40              parser.error('Ports must be integers.')
41
42          return host, int(port)
43
44      return peertype, parse_address(addresses)
```

Εικόνα II.5 *Ορισμός παραμέτρων εισόδου*

Το παράδειγμα της Εικόνας II.6 υποδεικνύει την υλοποίηση της κλάσης Peer η οποία κληρονομεί από την κλάση Protocol. Αν η εκτέλεση συμπεριφέρεται ως πελάτης, με το στήσιμο της σύνδεσης καλεί κάθε 5 λεπτά τη μέθοδο `sendUpdate`, η οποία στέλνει το string '<update>' στο διακομιστή. Αν η εκτέλεση συμπεριφέρεται ως διακομιστής, σε κάθε παραλαβή του '<update>' απαντά (μέθοδος `sendAck`) με μια επιβεβαίωση '<Ack>'.

```
47  class Peer(Protocol):
48
49      acks = 0
50      connected = False
51
52      def __init__(self, factory, peer_type):
53          self.pt = peer_type
54          self.factory = factory
55
56      def connectionMade(self):
57          if self.pt == 'client':
58              self.connected = True
59              reactor.callLater(5, self.sendUpdate)
60          else:
61              print "Connected from", self.transport.client
62              try:
63                  self.transport.write('<connection up>')
64              except Exception, e:
65                  print e.args[0]
66              self.ts = time.time()
67
68      def sendUpdate(self):
69          print "Sending update"
70          try:
71              self.transport.write('<update>')
72          except Exception, ex1:
73              print "Exception trying to send: ", ex1.args[0]
74          if self.connected == True:
75              reactor.callLater(5, self.sendUpdate)
76
77      def sendAck(self):
78          print "sendAck"
79          self.ts = time.time()
80          try:
81              self.transport.write('<Ack>')
82          except Exception, e:
83              print e.args[0]
84
85      def dataReceived(self, data):
86          if self.pt == 'client':
87              print 'Client received ' + data
88              self.acks += 1
89          else:
90              print 'Server received ' + data
91              self.sendAck()
92
93      def connectionLost(self, reason):
94          print "Disconnected"
95          if self.pt == 'client':
96              self.connected = False
97              self.done()
98
99      def done(self):
100         self.factory.finished(self.acks)
101
```

Εικόνα ΙΙ.6 *Κλάση Peer*

Το παράδειγμα της Εικόνας II.7 υποδεικνύει την υλοποίηση της κλάσης PeerFactory.

```
103 class PeerFactory(ClientFactory):
104
105     def __init__(self, peertype, fname):
106         print '@__init__'
107         self.pt = peertype
108         self.acks = 0
109         self.fname = fname
110         self.records = []
111
112     def finished(self, arg):
113         self.acks = arg
114         self.report()
115
116     def report(self):
117         print 'Received %d acks' % self.acks
118
119     def clientConnectionFailed(self, connector, reason):
120         print 'Failed to connect to:', connector.getDestination()
121         self.finished(0)
122
123     def clientConnectionLost(self, connector, reason):
124         print 'Lost connection.  Reason:', reason
125
126     def startFactory(self):
127         print "@startFactory"
128         if self.pt == 'server':
129             self.fp = open(self.fname, 'w+')
130
131     def stopFactory(self):
132         print "@stopFactory"
133         if self.pt == 'server':
134             self.fp.close()
135
136     def buildProtocol(self, addr):
137         print "@buildProtocol"
138         protocol = Peer(self, self.pt)
139         return protocol
140
```

Εικόνα II.7 *Κλάση PeerFactory*

Το παράδειγμα της Εικόνας II.8 αποτελεί το κυρίως πρόγραμμα, το οποίο διαχωρίζει τις περιπτώσεις κατά τις οποίες η εκτέλεση προορίζεται να λειτουργήσει ως πελάτης ή ως διακομιστής. Η λειτουργία πελάτη ξεκινά τη σύνδεση προς το διακομιστή και εισέρχεται στο βρόχο γεγονότων (event loop). Η λειτουργία διακομιστή ξεκινά να ακούει σε συγκεκριμένη πόρτα (στο παράδειγμα αυτό, στην

πόρτα 8888) και εισέρχεται στο βρόχο γεγονότων. Από το σημείο αυτό και μετά ο χειρισμός οποιουδήποτε γεγονότος εκτελείται ως η κλήση κάποιας συνάρτησης της κλάσης Peer.

```
__name__ == '__main__':
 peer_type, address = parse_args()


 if peer_type == 'server':
     factory = PeerFactory('server', 'log')
     reactor.listenTCP(8888, factory)
     print "Starting server @" + address[0] + " port " + str(add
 else:
     factory = PeerFactory('client', '')
     host, port = address
     print "Connecting to host " + host + " port " + str(port)
     reactor.connectTCP(host, port, factory)

 reactor.run()
```

Εικόνα II.8 *Κυρίως πρόγραμμα*

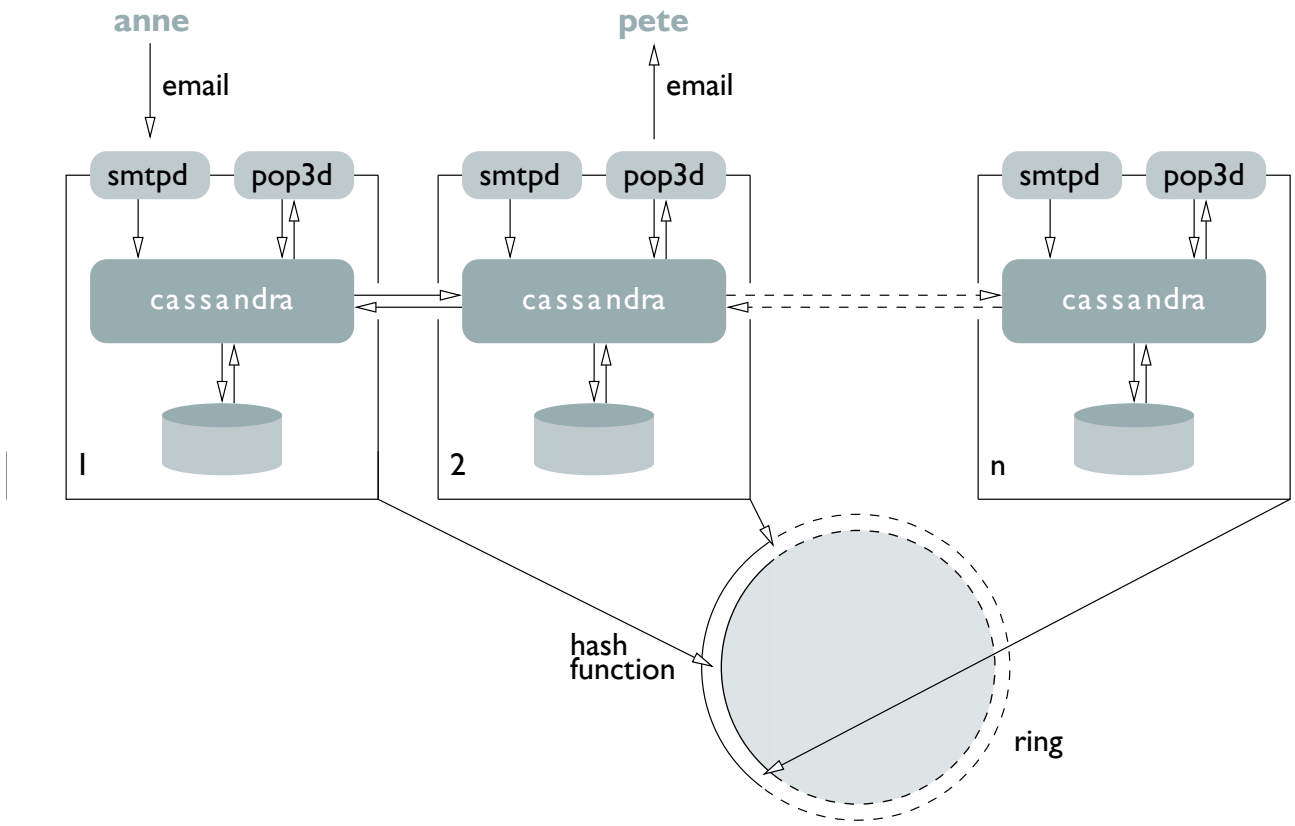
Ο αναγνώστης καλείται να πειραματιστεί με το πρόγραμμα των Εικόνων II.3-II.7, ξεκινώντας μια εκτέλεση διακομιστή σε ένα από τα τερματικά παράθυρα και μια εκτέλεση πελάτη σε ένα άλλο. Στη συνέχεια, η εκκίνηση ενός ακόμα πελάτη θα αναδείξει το πρόβλημα της διαχείρισης περισσότερης της μιας συνδέσεων στην πλευρά του διακομιστή (δείτε σχετική άσκηση στο τέλος του κεφαλαίου).

II.3 Προγραμματιστική πρόσβαση στο κατανεμημένο σύστημα Cassandra

Σε αυτήν την ενότητα εισάγουμε τον αναγνώστη στην προγραμματιστική πρόσβαση στο κατανεμημένο συστήματος αποθήκευσης Cassandra (Lakshman, A., & Malik, 2010). Η περιγραφή ακολουθεί την παρουσίαση επιστημονικής εργασίας των συγγραφέων (Koromilas & Magoutis, 2011) και εστιάζει στη συνοπτικότητα και ταχεία παραγωγή πρωτότυπου συστήματος που καθιστά δυνατή η χρήση της Python.

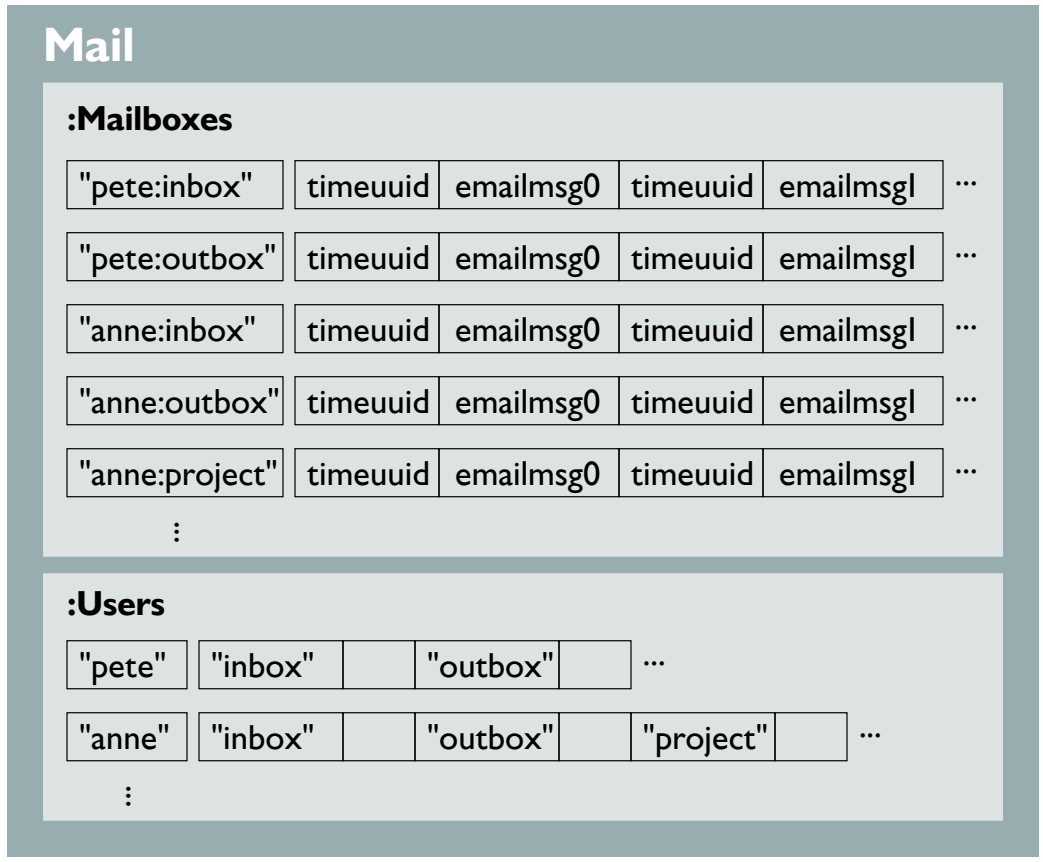
Το κατανεμημένο σύστημα αποθήκευσης Apache Cassandra ανήκει σε μια σχετικά νέα κατηγορία συστημάτων (γνωστή με το όνομα NoSQL) τα οποία προσφέρουν δομημένη οργάνωση δεδομένων, ταχεία αναζήτηση μέσω ευρετηρίων, χωρίς ωστόσο να υποστηρίζουν τις πλήρεις λειτουργίες διαχείρισης δεδομένων οι οποίες είναι διαθέσιμες στις παραδοσιακές σχεσιακές βάσεις δεδομένων SQL. Η νέα αυτή κατηγορία συστημάτων δημιουργήθηκε από την ανάγκη να επιτευχθεί καλύτερη κλιμακωσιμότητα από όση είναι εφικτή στις παραδοσιακές SQL παράλληλες/κατανεμημένες βάσεις δεδομένων, μειώνοντας τη λειτουργικότητα η οποία, ωστόσο, δεν είναι πάντα απαραίτητη στις εφαρμογές τις οποίες στοχεύουν τα NoSQL συστήματα.

Το σύστημα Cassandra, λοιπόν, αποτελείται από μια ομάδα κόμβων, λειτουργικά ισοδύναμων, που προσδίδουν αποθηκευτική ικανότητα στο σύστημα (Εικόνα II.9). Οι πελάτες του συστήματος (smtpd και pop3d στην Εικόνα II.9) μπορούν να συνδεθούν σε οποιοδήποτε κόμβο για να προσπελάσουν δεδομένα μέσω ενός απλού put/get API. Το ευρετήριο δεδομένων το οποίο χρησιμοποιεί το Cassandra, βασίζεται σε μια δομή hash η οποία υλοποιείται με μια συνάρτηση τύπου MD5 ή SHA-1 και απεικονίζει το κύριο κλειδί κάθε γραμμής δεδομένων ενός πίνακα σε έναν ή περισσότερους από τους κόμβους του συστήματος (περισσότερους του ενός όταν έχει επιλεχθεί η χρήση αντιγράφων για λόγους υψηλής απόδοσης και διαθεσιμότητας).



Εικόνα II.9 *Δομή κατανεμημένου συστήματος Cassandra και εφαρμογής SMTP server πάνω από αυτό*

Στη συνέχεια θα εστιάζουμε στο μοντέλο δεδομένων του Cassandra και το σχήμα δεδομένων που σχεδιάστηκε για την εφαρμογή διακομιστή E-Mail (πρωτόκολλο SMTP) πάνω από το Cassandra. Η συγκεκριμένη εφαρμογή θα αναφέρεται στη συνέχεια ως CassMail. Η βασική μονάδα δεδομένων του Cassandra είναι η στήλη (column) την οποία στη συνέχεια θα αναφέρουμε και ως block. Ένα block αποτελείται από ένα κλειδί (key) και μια τιμή (value). Μια ακολουθία από blocks (οποιοσδήποτε αριθμός από αυτά) μαζί αποτελούν μια γραμμή (row). Τα blocks σε μια γραμμή μπορούν να ταξινομηθούν με οποιοδήποτε τρόπο επιθυμεί ο χρήστης, με βάση τον τύπο του κλειδιού (για παράδειγμα, με χρονική σειρά). Κάθε γραμμή ταυτοποιείται από το δικό της ιδιαίτερο κλειδί. Μια γραμμή αντιστοιχίζεται σε έναν ή περισσότερους Cassandra κόμβους, βάσει της δομής hashing που αναφέραμε νωρίτερα. Οι γραμμές ομαδοποιούνται σε οικογένειες στηλών (column families), οι οποίες είναι έννοιες που μοιάζουν με σχεσιακούς πίνακες. Οι οικογένειες στηλών ομαδοποιούνται σε χώρους κλειδιών (keyspaces). Στην Εικόνα II.10 βλέπουμε το μοντέλο δεδομένων που σχεδιάστηκε για την αποθήκευση και αναζήτηση E-Mail από ένα διακομιστή του πρωτοκόλλου SMTP.



Εικόνα II.10 *Μοντέλο δεδομένων Cassandra για τον E-Mail (SMTP) εξυπηρετητή*

Το Cassandra, όπως τα περισσότερα αντίστοιχα κατανεμημένα συστήματα, συνοδεύονται από κάποιον πελάτη/οδηγό γραμμένο σε Python και ο οποίος προσφέρει μια προγραμματιστική διεπαφή (API) προς τις εφαρμογές. Σε αυτήν την ενότητα θα μελετήσουμε έναν τέτοιο πελάτη/οδηγό, το pycassa (pycassa 1.11.0 Documentation, 2015). Το παράδειγμα της Εικόνας II.10 υποδεικνύει τη χρήση, μεταξύ άλλων, του smtpd module, του re (regular expression), και του pycassa. Το smtpd υλοποιεί έναν SMTP server ως μια κλάση με τις παρακάτω μεθόδους:

- SMTPServer(localaddr, remoteaddr): Δημιουργεί ένα αντικείμενο τύπου SMTPServer στην τοπική πόρτα localaddr, σε σύνδεση με έναν upstream SMTP relay, στην πόρτα remoteaddr. Αυτή η κλάση κληρονομεί από το asyncore.dispatcher, ένα προγενέστερο του Twisted πλαίσιο ασύγχρονου δικτυακού προγραμματισμού, με αντίστοιχο βρόχο γεγονότων (asyncore.loop()).
- process_message(peer, mailfrom, rcpttos, data): Αυτή η μέθοδος πρέπει να γίνει override για να επεξεργαστεί ένα μήνυμα, διαφορετικά προκαλείται εξαίρεση τύπου NotImplementedError. Η διεύθυνση του απομακρυσμένου υπολογιστή είναι peer, η διεύθυνση του αποστολέα και πα-

ραλήπτη είναι mailfrom και rcpttos αντίστοιχα, και data είναι ένα string που περιέχει το μήνυμα του ηλεκτρονικού ταχυδρομείου (σε RFC 2822 format).

Το παράδειγμα της Εικόνας II.10 δημιουργεί έναν SMTP διακομιστή (CassSMTPServer) ο οποίος αποθηκεύει μηνύματα ηλεκτρονικού ταχυδρομείου στο Cassandra.

```
import smtpd
import asyncore
import time
import uuid
import pycassa
import re
import sys

class CassSMTPServer(smtpd.SMTPServer):
    def __init__(*args, **kwargs):
        smtpd.SMTPServer.__init__(*args, **kwargs)

    def process_message(self, peer, mailfrom, rcpttos, data):
        now = time.strftime('%a %b %d %T %Y', time.gmtime())
        head = 'From {0} {1}\n'.format(mailfrom, now)
        email = head + data
        user = re.search('(.*)@(.*)', rcpttos[0]).group(1)
        mbox = 'inbox'
        c = pycassa.ConnectionPool('Mail')
        cl = pycassa.cassandra.ttypes.ConsistencyLevel.ONE
        cf = pycassa.ColumnFamily(c, 'Mailboxes', write_consistency_level=c
        cf.insert('{0}:{1}'.format(user, mbox),
            {uuid.uuid1() : email})
        cf = pycassa.ColumnFamily(c, 'Users', write_consistency_level=cl)
        cf.insert(user, {mbox : ''})
        return


if __name__ == '__main__':
    port = int(sys.argv[1]) if len(sys.argv) > 1 else 8025
    s = CassSMTPServer(('0.0.0.0', port), None)
    asyncore.loop()
```

Εικόνα II.11 *Προγραμματιστική πρόσβαση πελάτη στο κατανεμημένο σύστημα Cassandra*

II.4 Επίλογος

Σε αυτό το κεφάλαιο εξετάσαμε παραδείγματα προγραμματισμού κατανεμημένων συστημάτων που ακολουθούν το μοντέλο πελάτη-διακομιστή (client-server). Ξεκινήσαμε με μια εισαγωγή στο προγραμματιστικό πλαίσιο Twisted, εξετάζοντας αρχικά τη δομή της κλάσης Echo και στη συνέχεια την κλάση Peer, η οποία μπορεί να ενσωματώσει λειτουργικότητα πελάτη ή/και διακομιστή, επιλέγοντας κατά περίπτωση δυναμικά με βάση τις παραμέτρους εισόδου. Επεκτείνοντας το παράδειγμα της κλάσης Peer, μπορεί κανείς εύκολα να δημιουργήσει πιο πλούσια λειτουργικότητα, όπως μηχανισμούς πολυεκπομπής (Άσκηση 2). Είδαμε, τέλος, ένα παράδειγμα προγραμματισμού της πλευράς του πελάτη στο κατανεμημένο σύστημα Cassandra, ένα δημοφιλές κατανεμημένο σύστημα αποθήκευσης ανοικτού πηγαίου κώδικα.

Βιβλιογραφία/Αναφορές

Birrell, A. D., & Nelson, B. J. (1983). Implementing Remote procedure calls. *ACM SIGOPS Operating Systems Review*, 17(5), 3.

Coulouris, G., Dollimore, J., & Kindberg, T. (2005). *Distributed systems: Concepts and Design* (4th Edition). Addison-Wesley Longman, Inc.

Koromilas, L., & Magoutis, K. (2011). CassMail: A scalable, highly-available, and rapidly-prototyped E-mail service. In *IFIP Distributed Applications and Interoperable Systems (DAIS 2011)* (pp. 278-291). Heidelberg: Springer.

Lakshman, A., & Malik, P. (2010). Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, *44*(2), 35–40.

McKellar, J. (2013). Twisted Python: The engine of your Internet. Retrieved from http://radar.oreilly.com/2013/04/twisted-python-the-engine-of-your-internet.html

pycassa 1.11.0 Documentation. (2015). Retrieved from https://pycassa.github.io/pycassa/

Twisted Matrix Labs. (2015). Retrieved from https://twistedmatrix.com

Wayner, P. (2015, March 30). 7 reasons why frameworks are the new programming

languages. *InfoWorld*. Retrieved from http://www.infoworld.com/article/2902242

Κριτήρια αξιολόγησης

Κριτήριο αξιολόγησης 1 (Βαθμός δυσκολίας: ●●●)

Ο αναγνώστης καλείται να ξεκινήσει μια εκτέλεση εξυπηρετητή Peer σε ένα από τα τερματικά παράθυρα και μια εκτέλεση πελάτη Peer σε ένα άλλο. Η εκκίνηση ενός ακόμα πελάτη αναδεικνύει την ανάγκη διαχείρισης περισσότερων της μιας συνδέσεων στην πλευρά του εξυπηρετητή, πρόβλημα το οποίο καλείστε να επιλύσετε.

Απάντηση/Λύση

Κριτήριο αξιολόγησης 2 (Βαθμός δυσκολίας: ●●●)

Η άσκηση αυτή σας ζητά να υλοποιήσετε ένα μηχανισμό πολυεκπομπής (multicasting) μεταξύ ενός αποστολέα και περισσοτέρων του ενός παραληπτών με χρήση του twisted. Η πολυεκπομπή είναι μια μορφή ομαδικής επικοινωνίας κατά την οποία κάθε μήνυμα που στέλνεται από τον αποστολέα λαμβάνεται από όλα τα μέλη της ομάδας των παραληπτών. Ο αποστολέας γνωρίζει εκ των προτέρων τον αριθμό των παραληπτών (έστω N) και περιμένει τις αιτήσεις συνδέσεών τους. Μετά τη δημιουργία των N συνδέσεων

προχωρά στην αποστολή ενός αριθμού μηνυμάτων, ενδεχομένως με κάποια χρονική καθυστέρηση μεταξύ τους, προς την ομάδα των παραληπτών.

Απάντηση/Λύση

Κριτήριο αξιολόγησης 3 (Βαθμός δυσκολίας: ●●●)

Εγκαταστήστε το Cassandra στο μηχάνημά σας, εκκινήστε το στην ελάχιστή του μορφή (με έναν κόμβο) και σε διαφορετικό τερματικό εκτελέστε το παράδειγμα της Εικόνας II.10. Έχοντας ξεκινήσει τον SMTP server, πειραματιστείτε με το συνολικό σύστημα, για παράδειγμα χρησιμοποιώντας ένα benchmark, όπως το Postal (http://doc.coker.com.au/projects/postal/).

Απάντηση/Λύση