RUDP API Explanation and Answers

```c
struct rudp_header {
    uint16_t length;
    uint16_t checksum;
    uint16_t flags;
};
```

This structure represents the header of a RUDP packet:

- `length`: Length of the payload (data) in the packet.
- `checksum`: A checksum value used for error detection.
- `flags`: Flags used to indicate the type of packet (e.g., `SYN`, `ACK`, `FIN`, `DATA`).

```c
uint16_t calculate_checksum(void *data, int length) {
    uint16_t *buf = (uint16_t *)data;
    unsigned long sum = 0;

    for (; length > 1; length -= 2) {
        sum += *buf++;
    }

    if (length == 1) {
        sum += *(uint8_t *)buf;
    }

    sum = (sum >> 16) + (sum & 0xFFFF);
    sum += (sum >> 16);

    return (uint16_t)(~sum);
}
```

Explanation:

- This function calculates a checksum for a block of data, which is a form of error detection.
- `sum += *buf++;`: Adds 16-bit chunks of data to the checksum.
- **Handling Odd Lengths**: If the length of the data is odd, the last byte is added separately.
- **Checksum Calculation**: The sum is folded by adding the carry and then taking the one's complement to create the final checksum.

```c
void rudp_send_packet(int sockfd, struct sockaddr_in *addr, socklen_t addr_size, char *data, uint16_t flags, double loss_percentage) {
    struct rudp_header header;
    char packet[BUFFER_SIZE];

    int data_len = strlen(data);
    if (data_len > BUFFER_SIZE - RUDP_HEADER_SIZE) {
        data_len = BUFFER_SIZE - RUDP_HEADER_SIZE;
    }

    header.length = htons(data_len);
    header.flags = htons(flags);
    header.checksum = 0;

    memcpy(packet, &header, RUDP_HEADER_SIZE);
    memcpy(packet + RUDP_HEADER_SIZE, data, data_len);

    header.checksum = calculate_checksum(packet, RUDP_HEADER_SIZE + data_len);
    memcpy(packet, &header, RUDP_HEADER_SIZE);

    // Simulate packet loss
    if ((rand() % 100) < (int)(loss_percentage * 100)) {
        return;
    }

    if (sendto(sockfd, packet, RUDP_HEADER_SIZE + data_len, 0, (struct sockaddr *)addr, addr_size) == -1) {
        error_handling("Failed to send packet");
    }
}
```

Explanation:

- `rudp_send_packet` is responsible for sending a packet using RUDP.
- **Packet Construction**:
  - The function constructs the packet by setting the header fields (length, flags, and checksum).
  - Data is copied into the packet after the header.
  - The checksum is calculated and added to the header.
- **Simulating Packet Loss**:
  - Before sending, the function checks if the packet should be "lost" by comparing a random number against the loss probability.
- **Sending the Packet**:
  - If the packet is not "lost," it is sent using `sendto`.

```c
ssize_t rudp_receive_packet(int sockfd, char *buffer, struct sockaddr_in *addr, socklen_t *addr_size, double loss_percentage) {
    char packet[BUFFER_SIZE];
    ssize_t bytes_received = recvfrom(sockfd, packet, BUFFER_SIZE, 0, (struct sockaddr *)addr, addr_size);

    if (bytes_received > 0) {
        // Simulate packet loss
        if ((rand() % 100) < (int)(loss_percentage * 100)) {
            return 0;  // Simulate as if the packet was not received
        }

        struct rudp_header *header = (struct rudp_header *)packet;
        uint16_t received_checksum = header->checksum;
        header->checksum = 0;
        uint16_t calculated_checksum = calculate_checksum(packet, bytes_received);

        if (received_checksum != calculated_checksum) {
            printf("Checksum error, packet discarded\n");
            return -1;
        }

        int data_length = ntohs(header->length);
        if (data_length > bytes_received - RUDP_HEADER_SIZE) {
            printf("Packet length error, packet discarded\n");
            return -1;
        }

        memcpy(buffer, packet + RUDP_HEADER_SIZE, data_length);
        return data_length;
    }

    return bytes_received;
}
```

Explanation:

- `rudp_receive_packet` is responsible for receiving and validating a RUDP packet.
- **Receiving the Packet**:
  - The function calls `recvfrom` to receive a packet.
- **Simulating Packet Loss**:
  - After receiving, it checks if the packet should be "lost" by comparing a random number against the loss probability. If it should be lost, the function returns 0, simulating a lost packet.
- **Checksum Verification**:
  - The received checksum is compared with the calculated checksum. If they don't match, the packet is discarded.
- **Data Extraction**:
  - The payload data is extracted from the packet and copied into the provided buffer.

1) על פי הנתונים שנאספו, TCP Cubic נתן תוצאות טובות יותר בסך הכל בהשוואה ל- TCP Reno. TCP Cubic, שנחשב למתקדם יותר, מצליח לנצל את רוחב הפס באופן יותר יעיל בסביבות עם הפסד חבילות נמוך, ומספק תוצאות טובות יותר גם תחת הפסד חבילות גבוה, בזכות האופן שבו הוא מתאושש מאיבוד חבילות. לעומת זאת, TCP Reno נוטה להיות שמרני יותר בהקצאת רוחב פס, ולכן בולטת חולשתו כאשר יש הפסד חבילות משמעותי, מה שגורם להורדת קצב הנתונים בתדירות גבוהה יותר.

מסקנות אלו מבוססות על ניתוח הנתונים, שמראה כי TCP Cubic מצליח לשמור על קצבים גבוהים יותר גם כשיש תנאים של הפסד חבילות.

2) ביצועי ה-Reliable UDP היו טובים יותר בהשוואה ל-TCP רגיל בתנאים של הפסד חבילות גבוה. Reliable UDP מתוכנן כך שהוא משדר מחדש חבילות שאבדו במהירות רבה יותר, בלי תלות בבקרה על עומס כמו ב-TCP, ולכן הוא מתאים יותר לתנאים בהם יש הפסד חבילות גבוה.

TCP, לעומת זאת, כפוף לאלגוריתמים של בקרה על עומס, מה שגורם לו להאט את קצב השידור בצורה משמעותית כאשר הוא מזהה איבוד חבילות. Reliable UDP מצליח להתמודד טוב יותר עם הפסדים בכך שהוא מבצע שידורים חוזרים מהירים יותר, מה שמוביל לביצועים טובים יותר בסביבה עם הפסד חבילות גבוה.

3)בהתאם לנתונים, נעדיף להשתמש ב-TCP במצבים בהם יש צורך בבקרה על עומס והגנה על רשתות מפני הצפה, כמו בשידורי וידאו חיים, גלישה באינטרנט, או כל יישום שבו יש צורך בוודאות שהנתונים יגיעו בצורה מסודרת ובזמן סביר, והפסד חבילות נמוך יחסית.

Reliable UDP, לעומת זאת, מתאים יותר ליישומים בהם הפסד חבילות גבוה יותר הוא שכיח והדרישה היא בעיקר לקצב שידור גבוה עם שידורים חוזרים מהירים, כגון בשידורים בזמן אמת של משחקים מקוונים או ביישומים צבאיים שבהם הנגישות למהירות ואמינות גוברת על הצורך בסדר המסירה של החבילות.

מסקנות אלו נובעות מהביצועים שנמדדו בתנאים שונים, אשר הצביעו על היתרונות והחסרונות של כל פרוטוקול בתנאים השונים.

השינוי של הגדלת ה-SSThreshold בתחילת הקשר ב-TCP עשוי להועיל במידה המרבית במקרה של קשר ארוך על גבי רשת אמינה עם RTT גדול (אופציה 1)

קשר ארוך: במקרה של קשר ארוך, יש הרבה נתונים להעביר, ולכן כל שיפור בתהליך ההתחלתי (כמו הגדלת ה-SSThreshold) ישפיע על הביצועים לאורך זמן. כמות הנתונים הגדולה מאפשרת לנצל את השינוי בצורה מרבית לאורך כל חיי הקשר.
רשת אמינה: ברשת שבה איבוד חבילות הוא נמוך, TCP יוכל לנצל באופן מלא את הגדלת ה-SSThreshold, מכיוון שהנתונים ישלחו בהצלחה עם פחות הפרעות והתאוששויות כתוצאה מאיבוד חבילות.
RTT גדול: ב-RTT גדול, זמן הסיבוב של חבילה ושל האישור שלה הוא ארוך. הגדלת ה-SSThreshold בתחילת הקשר תאפשר להגיע לקצב שידור גבוה יותר מהר יותר, מה שיפחית את זמן ההמתנה (שהוא בעייתי יותר ב-RTT גבוה) וישפר את ה-throughput לאורך זמן.
שילוב של קשר ארוך, רשת אמינה ו-RTT גדול הופך את המקרה הזה למועמד הטוב ביותר לניצול מיטבי של הגדלת ה-SSThreshold בתחילת הקשר.

הקשר מתחיל עם חלון בגודל MSS·1.

1. הקשר מסתיים כאשר חלון התעבורה מגיע ל-SSthresh, כלומר S·MSS.
2. במהלך הקשר לא אובדות חבילות.
3. SSthresh=S·MSS.
4. S·MSS<rwnd, כלומר חלון התעבורה של הלקוח מספיק גדול כדי להכיל את גודל החלון של ה-SSthresh.

שלב 1: זיהוי שלב ה- Slow Start: בשלב ה-Slow Start, חלון התעבורה מתחיל ב-MSS·1 ומכפיל את עצמו בכל RTT עד שמגיע ל-SSthresh. כלומר, החלון יגיע ל-S·MSS לאחר log2(S) RTTs, בהנחה שאין אובדן חבילות.

שלב 2: זיהוי שלב ה- Congestion Avoidance: לאחר שהגענו ל-SSthresh, אנחנו נכנסים לשלב ה-Congestion Avoidance, שבו גודל החלון גדל בקצב של MSS כל RTT, ולכן שיעור העלייה של החלון הוא קצב של MSS/RRT.

שלב 3: חישוב התפוקה: נחשב את התפוקה של הקשר.

- **בשלב ה-Slow Start**: התפוקה הממוצעת היא RTT1·MSS עבור הזמן שבו התעבורה היא MSS, 2·MSS·1, וכן הלאה. התפוקה הממוצעת יכולה להחשב כשיעור הסכום של התפוקות בקטע הזמן שבו גדל החלון.
- **בשלב ה-Congestion Avoidance**: במהלך השלב הזה, התפוקה הממוצעת תהיה בהתאם לקצב הצמיחה של חלון התעבורה. התפוקה הממוצעת תהיה בערך RTTS·MSS במשך הזמן שהקשר נמצא בשלב ה-Congestion Avoidance.

לסיכום: תשובה 3 היא הנכונה.