Chandransh Singh

22CS30017

KTP PROTOCOL IMPLEMENTATION DOCUMENTATION

TABLE OF CONTENTS

1. PROTOCOL OVERVIEW

KTP is a reliable transport protocol implementation that operates over UDP. It provides features similar to TCP including:

- Reliable data delivery through sequence numbers and acknowledgments
- Flow control through sliding window mechanism
- Ordered packet delivery

The protocol consists of a shared memory daemon (`initksocket`) that manages the underlying UDP sockets and protocol mechanics, plus a client API (`ksocket.c`) that applications use to send and receive data.

2. DATA STRUCTURES

`ktp_header_t` - Protocol message header

```
uint8_t  type;        // Message type (DATA or ACK)
uint8_t  seq_num;     // Sequence number
uint16_t rwnd;        // Receiver window size
uint8_t  last_ack;    // Last acknowledged sequence number
```

`ktp_message_t` - Complete protocol message

```
ktp_header_t header;           // Protocol control header
char data[KTP_MSG_SIZE];       // Message payload
```

`ktp_send_window_t` - Send window management

```
int size;                      // Current window size
uint8_t seq_nums[];            // In-flight sequence numbers
struct timeval send_times[];   // Packet transmission timestamps
int num_unacked;               // Count of unacknowledged packets
int base;                      // First unacknowledged packet index
int next_seq_num;              // Next sequence number to use
```

`ktp_recv_window_t` - Receive window management

```
int size;                      // Current window size (free slots)
uint8_t expected_seq_num;      // Next expected in-order sequence number
int buffer_occupied;           // Count of filled buffer slots
int buffer_read_pos;           // Next read position in buffer
int buffer_write_pos;          // Next write position in buffer
uint8_t received_msgs[];       // Received message tracking
int nospace_flag;              // Buffer full indicator
uint8_t last_ack_sent;         // Most recent ACK sequence number
```

`ktp_socket_t` - Main protocol socket structure

```
int is_allocated;              // Socket allocation status (1=allocated, 0=free)
pid_t pid;                     // Owner process ID
int udp_sockfd;                // Underlying UDP socket descriptor
int bind_requested;            // Binding request indicator
int is_bound;                  // Bound state indicator
struct sockaddr_in src_addr;   // Local address
struct sockaddr_in dst_addr;   // Remote address
ktp_send_window_t swnd;        // Send window
ktp_recv_window_t rwnd;        // Receive window
char send_buffer[][KTP_MSG_SIZE]; // Outgoing message buffer
int send_buffer_occ[];         // Send buffer occupation flags
char recv_buffer[][KTP_MSG_SIZE]; // Incoming message buffer
pthread_mutex_t socket_mutex;  // Socket access lock
```

3. CONSTANTS

Socket and Buffer Configuration

```
#define KTP_MAX_SOCKETS 10        // Maximum concurrent KTP sockets
#define KTP_MSG_SIZE 512          // Fixed message payload size
#define KTP_RECV_BUFFER_SIZE 10   // Receive buffer capacity (packets)
#define KTP_SEND_BUFFER_SIZE 10   // Send buffer capacity (packets)
#define KTP_MAX_WINDOW_SIZE 10    // Maximum sliding window size
```

Protocol Parameters

```
#define KTP_TIMEOUT_SEC 5         // Retransmission timeout in seconds
#define KTP_PACKET_LOSS_PROB 0.15 // Simulated packet loss probability
```

Socket Types

```
#define SOCK_KTP 1000             // KTP socket type identifier
```

4. Global Variables

Client API (`ksocket.c`)

```
ktp_socket_t* ktpSocketArray;
pthread_mutex_t globalMutex;
```

Protocol Daemon (`initksocket.c`)

```
int isRunning;
int shmHandle;
ktp_socket_t* ktpSockets;
```

5. Error Codes

Protocol-specific Error Codes

```
#define E_KTP_NO_SPACE    1001
#define E_KTP_NOT_BOUND   1002
#define E_KTP_NO_MESSAGE  1003
```

Standard System Error Codes Used

```
EINVAL, EBADF, ETIMEDOUT
```

6. CLIENT API FUNCTIONS

6.1 SHARED MEMORY ACCESS

`get_ktp_sockets()`

- **Purpose:** Accesses shared memory containing KTP socket structures
- **Parameters:** None
- **Returns:** Pointer to KTP socket array, `NULL` if error occurs
- **Details:**
  - Uses thread-safe double-checking pattern with mutex
  - Attaches to existing shared memory segment created by daemon
  - Generates consistent key based on `/tmp` and `'K'`
  - Provides inter-process access to socket structures

6.2 SOCKET LIFECYCLE MANAGEMENT

`k_socket(int domain, int type, int protocol)`

- **Purpose:** Creates new KTP socket
- **Parameters:**
  - `domain` : Address family (`AF_INET`)
  - `type` : Must be `SOCK_KTP`
  - `protocol` : Usually `0`
- **Returns:** Socket descriptor (≥0) on success, `-1` on failure
- **Details:**
  - Validates socket type is `SOCK_KTP`
  - Finds available slot in shared memory array
  - Initializes send/receive windows with default values
  - Returns array index as socket descriptor

`k_close(int sockfd)`

- **Purpose:** Releases KTP socket resources
- **Parameters:**
  - `sockfd` : Socket descriptor to close
- **Returns:** `0` on success, `-1` on failure
- **Details:**

- Marks socket as unallocated
- Clears send and receive buffers
- Resets window structures and tracking information
- Prepares socket slot for reuse

6.3 CONNECTION ESTABLISHMENT

`k_bind(int sockfd, const char* src_ip, int src_port, const char* dst_ip, int dst_port)`

- **Purpose:** Associates socket with network endpoints
- **Parameters:**
  - `sockfd` : Socket descriptor
  - `src_ip` : Source IP address string
  - `src_port` : Source port number
  - `dst_ip` : Destination IP address string
  - `dst_port` : Destination port number
- **Returns:** `0` on success, `-1` on failure
- **Details:**
  - Configures source and destination addressing
  - Sets `bind_requested` flag for daemon processing
  - Waits with timeout for binding completion
  - Binding performed by receiver thread in daemon

6.4 DATA TRANSFER OPERATIONS

`k_sendto(int sockfd, const void *buf, size_t len, int flags, const struct sockaddr *dest_addr, socklen_t`

- **Purpose:** Queues data for transmission
- **Parameters:**
  - `sockfd` : Socket descriptor
  - `buf` : Data buffer
  - `len` : Data length
  - `flags` : Currently unused
  - `dest_addr` : Destination address
  - `addrlen` : Address structure length
- **Returns:** Bytes queued on success, `-1` on failure
- **Details:**
  - Validates socket is bound and destination matches
  - Finds available slot in send buffer
  - Copies data into shared buffer
  - Marks buffer position as occupied for sender thread
  - Actual transmission handled by daemon

`k_recvfrom(int sockfd, void *buf, size_t len, int flags, struct sockaddr *src_addr, socklen_t *addrlen)`

- **Purpose:** Retrieves next available message
- **Parameters:**
  - `sockfd` : Socket descriptor
  - `buf` : Buffer to store data
  - `len` : Maximum bytes to read
  - `flags` : Currently unused
  - `src_addr` : Optional source address storage
  - `addrlen` : Optional address length pointer
- **Returns:** Bytes received on success, `-1` on failure
- **Details:**
  - Checks for available messages in receive buffer
  - Copies data from shared buffer to user buffer
  - Updates buffer state and flow control parameters
  - Returns `E_KTP_NO_MESSAGE` when buffer empty

6.5 PROTOCOL TESTING

`dropMessage(float p)`

- **Purpose:** Simulates network packet loss
- **Parameters:**
  - `p` : Loss probability (`0.0`–`1.0`)
- **Returns:** `1` if packet should drop, `0` if deliver
- **Details:**
  - Validates probability is in valid range
  - Generates random number for comparison
  - Used for testing reliability mechanisms

7. DAEMON IMPLEMENTATION

7.1 CORE FUNCTIONS

`init_shared_memory()`

- **Purpose:** Creates and initializes shared memory segment
- **Returns:** `0` on success, `-1` on failure
- **Details:**
  - Generates key using `ftok()`
  - Creates shared memory segment accessible to all processes
  - Pre-creates UDP sockets for all slots
  - Initializes process-shared mutexes
  - Prepares socket structures for client applications

`cleanup()`

- **Purpose:** Releases all allocated resources
- **Details:**
  - Detaches from shared memory
  - Removes shared memory segment
  - Called during normal shutdown or error conditions

`handle_signal(int sig)`

- **Purpose:** Handles termination signals
- **Parameters:**
  - `sig` : Signal number received
- **Details:**
  - Sets running flag to `0` to trigger graceful shutdown
  - Allows threads to complete current operations

`main()`

- **Purpose:** Daemon entry point and lifecycle manager
- **Details:**
  - Sets up signal handlers (`SIGINT` , `SIGTERM`)
  - Initializes random number generator
  - Creates shared memory
  - Launches three protocol threads
  - Waits for termination signal
  - Ensures clean shutdown of all resources

8. PROTOCOL OPERATION DETAILS

8.1 CONNECTION MANAGEMENT

Binding Process:

- Client initiates: Calls `k_bind()` with endpoint information

- Request registration: Sets `bind_requested` flag in socket structure
- Daemon processing: Receiver thread detects flag and binds UDP socket
- Synchronization: Client waits with timeout for completion
- Completion: `is_bound` flag set when successful

8.2 DATA TRANSFER MECHANISMS

Transmission Flow:

1. Client queues message via `k_sendto()`
2. Sender thread discovers queued message
3. Protocol header added with sequence number
4. Message transmitted via underlying UDP socket
5. Transmission record kept for reliability tracking
6. Window state updated to reflect in-flight packet

Reception Flow:

1. Receiver thread detects incoming UDP packet
2. Message validated (duplicate check, window constraints)
3. Valid messages stored in receive buffer
4. In-order messages advance expected sequence number
5. Acknowledgment sent with current window state
6. Client retrieves message via `k_recvfrom()`

8.3 RELIABILITY MECHANISMS

- **Packet Loss Handling:** Timeout-based detection and retransmission
- **Duplicate Detection:** Sequence number tracking in receive window
- **Retransmission Strategy:** Go-back-N approach
- **Timeout Period:** `KTP_TIMEOUT_SEC` (5 seconds)

8.4 FLOW CONTROL

- **Window Management:** Receiver advertises available buffer space in ACKs
- **Congestion Handling:** Zero window handling, buffer-full detection

8.5 PROTOCOL EFFICIENCY

- **ACK Strategy:** Cumulative acknowledgments
- **Buffer Management:** Circular buffer implementation for efficiency

This protocol provides reliable, in-order message delivery despite network unreliability, enabling effective communication between distributed processes.

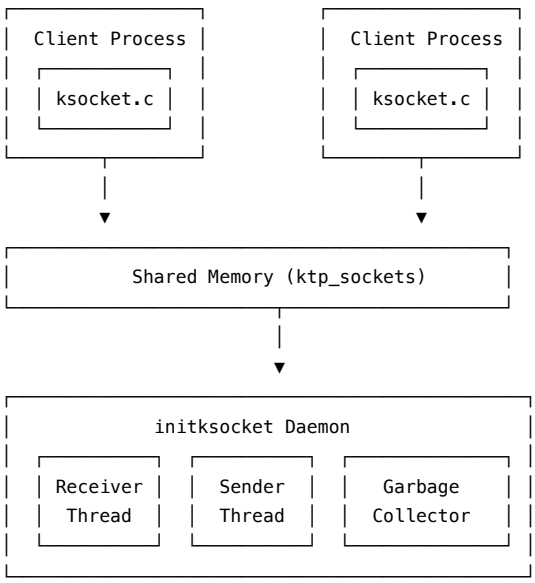
9. PACKET TRANSMISSION STATISTICS

Drop Probability	Packets in File	Packets Transmitted	Avg Transmissions per Packet	Total Packets	Dropped Packets	Drop Percentage
0.00	207	208	1.00	616	0	0.00%
0.05	207	266	1.28	676	39	5.77%
0.10	207	321	1.55	709	70	9.87%
0.15	207	386	1.86	763	131	17.17%
0.20	207	389	1.88	752	146	19.41%
0.25	207	439	2.12	821	188	22.90%
0.30	207	460	2.22	827	248	29.99%
0.35	207	589	2.85	1040	364	35.00%
0.40	207	601	2.90	1031	408	39.57%

Drop Probability	Packets in File	Packets Transmitted	Avg Transmissions per Packet	Total Packets	Dropped Packets	Drop Percentage
0.45	207	700	3.38	1139	509	44.69%
0.50	207	821	3.97	1288	650	50.47%

> File size: 105334B ≈ 102KB
> Note: One additional packet was send for metadata

10. KTP PROTOCOL ARCHITECTURE OVERVIEW



11. Running the KTP Protocol

Follow these steps to run the KTP protocol for reliable file transfer:

1. **Compile the project**

   ```
   make
   ```

2. **Start the KTP daemon** in one terminal:

   ```
   ./initksocket
   ```

3. **Start the receiver (`user2`)** in another terminal:

   ```
   ./user2 <src_ip> <src_port> <dst_ip> <dst_port> <output_filename>
   ```

   eg: `./user2 127.0.0.1 5055 127.0.0.1 5056 out.txt`
4. **Start the sender (`user1`)** in another terminal:

   ```
   ./user1 <src_ip> <src_port> <dst_ip> <dst_port> <input_filename>
   ```

   eg: `./user1 127.0.0.1 5056 127.0.0.1 5055 input.txt`
5. **Wait for the file transfer to complete**
   - Monitor the `user2` terminal until the transfer finishes.
6. **Terminate the KTP daemon**
   - Once the transfer is complete, stop `initksocket` by pressing `Ctrl + C`