Overview of TCP Transport Modifications to WireGuard

This document summarizes the modifications made to integrate TCP transport into the WireGuard VPN protocol, specifically focusing on the initialization, socket management, handshake processing, and data transfer.

Tunneling a VPN over TCP encounters several obstacles to overcome:

- ***Triple Socket Peer/Device State Engines:*** The TCP connection oriented protocol encounters several tricky states with connection initiation and handshake exchange combined with connection setup signaling and establishment that can derail the WireGuard Noise Protocol handshakes.

  We have solved this problem by employing a novel three+ connection system for WireGuard peers, we maintain both inbound and outbound sockets potentially active with TCP state and handshake exchanges active simultaneously. We keep timestamps of inbound and outbound activity/connections and handshake exchanges to bias traffic towards the inbound/outbound link that the last active handshake completed on. But since the other side may be initiating a connection right when we mark a handshake link active, we do not tear down the link even if we are using the outbound link, we essentially maintain a double state engine for every peer.

  The other part of this problem is supporting roaming and address changes or allowing third link be established while our dual inbound/outbound state engine. So the third connection we support is a list of pending tcp connections maintained by potentially two listeners - for IPv4 and IPv6 - that handle new inbound connections. But to prevent folks being able to DoS the services with connection blasters, we don't just blindly switch to the new connection. We add a list to the device called the `tcp_connection_list` which maintains a list of temporary peer structures for these inbound connections.

  We needed the temporary peer structures with their socket state engines for each connection that will enable the socket packet reception for each of these connections. (more on this below). But inbound connections aren't really valid until their handshake initiation message is processed and the WireGuard noise protocol responds with a handshake response and initiate the keypair decryption. So we idle the inbound connections in this list until the upper layers of the WireGuard Noise Protocol validates it and responds back to the valid peer - in the WireGuard system, to avoid information leakage, the protocol is silent for invalid credentials. When the outbound packet is sent out again we promote the connection to be the real peer outbound, and remove it from the pending list and free the temporary peer. A connection cleanup worker tasks wakes up periodically and flushes the pending list, to free any kernel resources (memory, queues, scheduled tasks) each connection uses.

- ***UDP to TCP Encapsulation and Header Creation/Stripping:*** To read and write UDP packet over a byte stream bidirectional TCP socket, we use a novel design for an encapsulation protocol and a system efficient set of transfer mechanism that are tightly integrated into the kernel TCP engine directly, which dispatches kernel work queue requests to transfer the packets.

  The packets read/and written to the socket asynchronously, i.e. TCP packets can hold partial parts of packets or multiple packets in transport. We give the packets to the TCP subsystem byte stream by stripping the network and UDP headers and taking the inner payload packet and header, but we process that with our own eight byte encapsulation header with a high speed checksum calculation that protects it. This encapsulation header lets the receive engine frame align received packets in a potentially desynchronized byte stream. The encapsulation header lets the receiver validate the length and delineation of the transmitted payloads over the byte stream.

  If the stream gets desynchronized, the de-encapsulation engine will read byte by byte until it sees another valid encapsulation header checksum to resynchronize to the inbound stream of encrypted packets again. On the receive side once encapsulation header validates, we recreate the UDP and network headers we stripped out on the sender end from the socket and peer context - we need to recreate these headers accompanying for the remainder of the WireGuard processing, and eventual resubmission of the packet/skb to the UDP tunnel interface WireGuard uses as its main interface to the linux networking stack.

- ***Efficient TCP processing directly wired in the kernel TCP stack:*** There is a transfer queue that transfers packets from the transmit queue to our encapsulation engine, and queuing up to be transmitted to the stack. To do this efficiently, we interject our processing directly into the kernel socket Data Ready state that is set whenever packets are received , and similarly the kernel socket Write Space (available) settings call back routines inside the kernels stack. We also similarly intercept the TCP State Change engine and track the connection state directly from kernel callback indications. For each connection the peer data structure stores the original addresses of original stack callback handler functions of each inbound and outbound socket and we interject our handlers for them instead. Our handlers then jump to the original code after we have taken care of our processing to track state or schedule workers to start for either reception or transmission.

  By using this direct connection from the WireGuard decryption engine and normal high speed and efficient packet processing we avoid some of the typical buffering elasticity typically encountered with tunneling tcp within tcp and navigating the user<>kernel boundary multiple times in userland VPN solutions - minimizing data copies and data handling.

- ***Near Future Enhancements and Performance Measurements*** - in the short term there are two enhancements we intend to deploy to this code:

    1. *Enhanced remote state signaling*: using the TCP encapsulation protocol - currently a shortcoming of our system is that the remote listener configuration is static. While the system accommodates dynamic ip address changes for the remote end i.e. roaming mobile clients, and dynamic connection endpoint ports fro the sockets, the remote listener port, i.e. the static WireGuard device port, typically configured to be 51820 must be configured to the same endpoint port remotely as the local listener, and modifying this configuration require tearing down the tunnel and reconfiguring the device. We have a design enhancement that will enable the remote end to signal its listener configuration along with a handshake packet, that will allow dynamic reconfiguration of this part of the tunnel configuration dynamically, further reducing the fingerprintability and allowing dynamic reconfiguration of all aspects of the encrypted tunneling.

    2. *Performance valuation and an initial state tuning* on the aggressiveness of the scheduling and kernel CPU use for de-encapsulation engine transfer to the WireGuard regular receive logic to optimize buffer memory usage. In general there are a large number of tunable performance parameters, and there is likely a whole PdD dissertation on optimal performance adjustment for this kind of tunneling system in response to different traffic and protocol mixes, but we will do an initial evaluation and simple optimizations.

       There are future enhancements involving MTU signaling and many other aspects of the operations between the two ends of the socket and various adjustments that can be considered for further future research and optimization.

    3. There are some errors that prevent module loading and reloading. Some resource deallocation is left dangling and we will clean that up.

The complete form and sequence of processing in our modified module follows this pattern:

1. Initialization and Setup:

- The module initialization (`wg_mod_init`) completes successfully, setting up necessary global structures, including `wg_genetlink_init`.
- Network interfaces are identified, and default IPv4 interfaces are configured for WireGuard operations.

2. Device and Peer Setup:

- The WireGuard device is set up using `wg_setup` and linked via `wg_newlink`. This involves initializing allowed IPs (`wg_allowedips_init`), setting up public key hashtables (`wg_pubkey_hashtable_alloc`), and configuring packet queues and workers for multicore processing (`wg_packet_queue_init`).
- Peer-specific structures are initialized, including cookie checkers, ratelimiters, and timers (`wg_cookie_checker_init`, `wg_ratelimiter_init`, `wg_timers_init`).

3. TCP Listener and Connection Management:

- A TCP listener is established using `wg_tcp_listener_socket_init`. It involves creating and binding IPv4 sockets, setting socket options, and starting listener threads.
- The listener accepts incoming connections (`wg_tcp_listener_worker`), creating temporary peers for handling inbound TCP connections. The `wg_add_tcp_socket_to_list` function manages these new socket connections.
- TCP-specific callbacks are set up to manage events such as state changes and data readiness (`wg_tcp_state_change`, `wg_tcp_data_ready`).

4. TCP Connection Establishment:

- TCP connections are initiated using the `wg_tcp_connect` function. This involves setting peer endpoints, creating outbound sockets, and assigning appropriate callbacks for connection management.
- The state of each peer's connection (established, pending) is tracked, with diagnostic information logged (`wg_peer: tcp_established`, `wg_peer: tcp_pending`).

5. Handshake and Session Management:

- The handshake process, integral to WireGuard, is adapted for TCP. Initiation and response packets are processed with TCP-specific handling (`wg_packet_handshake_send_worker`, `wg_receive_handshake_packet`).

- Ephemeral keys are generated and used in handshakes, with cryptographic operations like HMAC and KDF being performed to secure the session (`wg_noise_handshake_create_initiation`, `wg_noise_handshake_consume_response`).
- The function `wg_noise_handshake_begin_session` marks the transition from handshake to established session, with derived keys being inserted into hashtables.

6. Data Transmission:

- Data packets are prepared and sent over established TCP connections. Functions like `wg_tcp_write_worker` and `wg_tcp_send` handle the encapsulation of WireGuard payloads within TCP segments.
- Packet integrity is verified using checksum functions, ensuring that data is correctly encapsulated and transmitted (`wg_header_checksum`).

7. Error Handling and Diagnostics:

- Logs capture detailed diagnostic information, including peer state, socket status, and handshake progression. This data is vital for troubleshooting and ensuring the robustness of the TCP transport layer.
- Error conditions, such as invalid address families or socket failures, are identified and managed appropriately, allowing for graceful degradation or recovery (`wg_tcp_state_change`, `wg_peer_put`).

Summary

These modifications allow WireGuard to operate over TCP, enabling compatibility with environments where UDP transport may be restricted. By adapting the handshake, session management, and data transmission routines to function over TCP, these changes maintain the security and efficiency characteristic of WireGuard while offering greater flexibility in network deployment scenarios.

New Structures and Global Variables

- **`struct wg_tcp_socket_list_entry`**: Manages a list of TCP sockets and related metadata for TCP connections.
- **`struct wg_socket_data`**: Holds device and peer pointers, indicating if a connection is inbound, used in setting socket user data.
- **`struct default_interface_info`**: Stores default network interface information (IPv4 and IPv6), which may be used for determining the default route and source addresses.
- These structures and global variables introduce additional state and tracking mechanisms for handling TCP connections, which are absent in regular UDP-based WireGuard.

New Functions for Endpoint Management and TCP Setup

- **`wg_setup_tcp_socket_callbacks()`** and **`wg_reset_tcp_socket_callbacks()`**: Manage socket callbacks, enabling and disabling data readiness and other socket-related event handling.
- **`wg_get_endpoint_from_socket()`**: Extracts endpoint information from a socket. This function is essential for handling connections dynamically based on socket states.
- **`log_wireguard_endpoint()`**: Logs endpoint information, indicating more complex endpoint management compared to regular WireGuard.
- These functions introduce the management of stateful TCP connections, handling different scenarios where peers may connect or reconnect using TCP.

Modified Transmission Functions

- **`send4()`** and **`send6()`**: These functions are responsible for sending data over IPv4 and IPv6, respectively. They are extended to work with TCP by adjusting the handling of routes and destinations, employing `udp_tunnel_xmit_skb()` for encapsulation, and setting flags for handling socket buffer (`skb`) operations.
- Modifications to these functions show an adaptation to handle the TCP transmission path by considering endpoint states, potentially sharing logic with UDP but diverging where TCP-specific behaviors are necessary.

TCP Worker Functions

- **`wg_tcp_transfer_worker()`**: Handles the transmission of data over TCP. It processes sk_buff items, routes them via the appropriate TCP queue, and manages the state and cleanup of work items. This function directly interacts with the TCP state, scheduling and descheduling TCP data transmission workers, which is an extension not seen in the UDP-only WireGuard.

Listener Management and Peer Creation

- **`wg_tcp_listener_worker()`**, **`wg_tcp_listener4_thread()`**, **`wg_tcp_listener6_thread()`**: These functions implement the logic for managing incoming TCP connections. They create listening threads, accept new connections, and manage temporary peers, extending the functionality to handle roaming and dynamically changing peer addresses.
- **`wg_temp_peer_create()`** and **`wg_add_tcp_socket_to_list()`**: These functions support temporary peer creation for handling new TCP connections and maintaining a list of active TCP sockets.
- These segments introduce the mechanics required for managing TCP listeners and ensuring that WireGuard can accept and handle new connections dynamically, setting up temporary peers and transitioning them into fully recognized connections based on handshake and endpoint validation.

Socket Initialization and Reinitialization

- **`wg_socket_init()`**: Initializes sockets with a TCP-specific configuration, leveraging the `udp_tunnel_sock_cfg` structure to set up the socket appropriately.
- **`wg_socket_reinit()`**: Reinitializes sockets, incorporating synchronization with existing TCP states and ensuring that the old socket resources are properly cleaned up.
- These changes expand the socket management to handle both TCP and UDP, requiring coordination and careful handling of state transitions to maintain secure and reliable communications.

TCP-Specific Cleanup and Timeout Management

- **`wg_clean_peer_socket()`**: Handles the cleanup of TCP socket states, releasing resources and managing pending connections, read/write workers, and workqueues associated with TCP transmission.
- **`wg_set_socket_timeouts()`**: Configures send and receive timeouts for TCP sockets, critical for managing the connection lifecycle and preventing resource exhaustion due to hanging TCP sessions.

Endpoint and Peer Matching

- **`wg_find_peer_by_endpoints()`**: Searches for peers by matching TCP endpoints, allowing for roaming and reconnection scenarios.
- **`endpoint_eq()`** and **`wg_endpoints_match()`**: Functions for comparing endpoints, critical in managing both the static and dynamic states of peers and their respective endpoints.

TCP Listener Initialization (`wg_tcp_listener_socket_init`):

- Checks to ensure proper initialization of both IPv4 and IPv6 listener sockets.
- The routine now uses `default_iface_info` to configure listeners for both IPv4 and IPv6 if available, setting up `tcp_listen_socket4` and `tcp_listen_socket6` respectively.
- Initiates the listener threads for both IPv4 and IPv6 only if the configuration and state of the device support it.
- Schedules a cleanup worker to handle stale connections periodically, ensuring system stability.

TCP Connection Establishment (`wg_tcp_connect`):

- Ensures that a TCP connection is only attempted if the peer's device transport is set to TCP and no existing connection is established or pending.
- Implements logic to set up source and destination addresses for both IPv4 and IPv6 using the `default_iface_info`.
- Handles socket creation and binding, implementing non-blocking connect calls and setting up appropriate callbacks.
- Allocates `wg_socket_data` for the socket and stores necessary peer information for subsequent handling.

TCP Connection Release (`wg_release_peer_tcp_connection`):

- Safely resets socket callbacks and performs a graceful shutdown of the socket.
- Handles the cleanup of associated resources, including SKB queues, scheduled work, and diagnostic data.
- Clears TCP connection flags and ensures proper state resetting for future connection attempts.

TCP State Change Handling (`wg_tcp_state_change`):

- Tracks state changes and errors.

- Handles different TCP states, including connection establishment, closure, and error handling.
- Implements logic to identify inbound vs. outbound connections and adjust peer state accordingly.
- Schedules appropriate work for connection retries and socket removal based on the detected state.

Packet Queuing (`wg_tcp_queuepkt`):

- Ensures packets are only queued if a valid TCP connection is established.
- Handles sending logic using `wg_tcp_send` function, retrying connection setup if the connection is not ready.
- Manages packet queues and triggers sending based on socket writeability

TCP Send Routine (`wg_tcp_send`):

- Encapsulates data with a TCP-specific header, including length, type, flags, and checksum.
- Uses `kernel_sendmsg` to send both header and payload efficiently.
- Provides detailed error handling and diagnostic logging for troubleshooting transmission issues.

TCP Read Worker (`wg_tcp_read_worker`):

- Handles reading from the socket in a non-blocking manner.
- Manages partial reads and synchronization of TCP encapsulation headers, ensuring the integrity of data streams.
- Implements buffering logic to accumulate complete packets before processing.

Callback Setup and Reset (`wg_setup_tcp_socket_callbacks`, `wg_reset_tcp_socket_callbacks`):

- Dynamically assigns and resets socket callbacks for state changes, write space, and data readiness.
- Uses `wg_socket_data` to track the peer and device associations with each socket.
- Ensures the callbacks are restored to their original state when the socket is released or reconfigured.

Connection List Management (`wg_add_tcp_socket_to_list`, `wg_remove_from_tcp_connection_list`, `wg_destruct_tcp_connection_list`):

- Manages a list of active TCP connections associated with a WireGuard device.

- Adds entries upon new connection establishment and removes stale or terminated connections.
- Uses RCU (Read-Copy-Update) for safe concurrent access and modifications, enhancing performance and safety.

TCP Cleanup and Retry Logic (`wg_tcp_cleanup_worker`, `wg_tcp_retry_worker`):

- Periodic cleanup of inactive TCP connections to free resources.
- Handles retries for outbound TCP connections, attempting reconnection after failures.
- Utilizes delayed work to schedule cleanup and retry tasks efficiently.

Temporary Peer Management (`wg_temp_peer_create`):

- Creates temporary peer structures for handling incoming connections.
- Initializes necessary fields for TCP management, including connection state flags and SKB queues.
- Allocates workqueues for managing asynchronous TCP read and write operations.

Updating Peer Endpoints and Promoting Connections:

- Modifications in `wg_receive_handshake_packet()` handle TCP-specific logic. Additional logic is added in the handshake reception logic to update the endpoint for TCP connections and promote a connection from the pending list to active for successful handshakes:

```
    if (!endpoint_eq(&peer->endpoint, &ep) && peer->device->transport
    == WG_TRANSPORT_TCP) {
  // TCP-specific handling
  if (socket_iter->temp_peer && endpoint_eq(&peer->endpoint,
&socket_iter->temp_peer->endpoint)) {
      // Promote connection from pending list
  }
}
```

  This code checks if the current connection matches an entry in the pending list and, if so, promotes it to an active connection, updating the peer's state accordingly.

Protection for TCP-Related State:

- Spinlocks are used to protect TCP-related state changes and queues:

```
spin_lock_init(&peer->tcp_lock);  // Lock for protecting TCP
state
```

```
spin_lock_init(&peer->send_queue_lock);  // Lock for protecting
the TX send queue
```

These locks ensure that modifications to TCP state and queues are thread-safe, preventing race conditions and inconsistent states.

Peer Cleanup and Removal Functions:

- The `wg_peer_remove()` and `wg_peer_remove_all()` functions are modified to ensure TCP resources are cleaned up when a peer is removed:

  ```
  wg_clean_peer_socket(peer, true, true, false); // Clean up both
  inbound and outbound TCP sockets

  peer_make_dead(peer);  // Mark the peer as dead and clean up
  resources
  ```

- These functions call `wg_clean_peer_socket()` to handle the removal of TCP connections and associated resources, ensuring that no lingering TCP connections remain after a peer is removed.

Overview of TCP-Related Receiver Modifications

1. **TCP Transport Support Definition:**

A new definition is added to differentiate between UDP and TCP transport:

```
#define WG_TRANSPORT_UDP    0
#define WG_TRANSPORT_TCP    1
```

2. **TCP Socket List Entry Structure:**

A new structure `wg_tcp_socket_list_entry` is introduced to manage per WireGuard device pending TCP connections - typically from roaming clients:

```
struct wg_tcp_socket_list_entry {
    struct socket *tcp_socket;        // Socket associated with the
connection
    struct sockaddr_storage src_addr; // Source address for the
connection
    struct wg_peer *temp_peer;        // Temporary peer for data-ready
callbacks
    struct list_head tcp_connection_ll;     // Linked list pointer
    ktime_t timestamp;                // Timestamp for connection
management
};
```

   - This structure holds information about a TCP connection, including the socket, source address, associated temporary peer, and a linked list pointer for managing connections.

3. **Socket Data Structure:**

The `wg_socket_data` structure includes a boolean to indicate if a connection is inbound. This data structure is attached to the `sk->sk_user_data`: context pointer used with each of the socket connections we initiate or the receiver listener processes, so that the rest of our system can associate any socket parameter with the device and peer it is used for when passed as a parameter

```
struct wg_socket_data {
    struct wg_device *device;
    struct wg_peer *peer;
    bool inbound;
};
```

Modifications to `wg_peer` Structure:

1. **TCP Socket Management:**
    - `struct socket *peer_socket, *inbound_socket, *outbound_socket;`
        - These pointers store the main peer socket, as well as separate inbound and outbound socket pointers for distinguishing connection directions.
        - `peer_socket` is the active socket used for communication.
        - `inbound_socket` handles incoming connections.
        - `outbound_socket` handles outgoing connections.

2. **Callback Function Pointers:**
    - ```
      void (*original_outbound_state_change)(struct sock *sk);
      void (*original_outbound_write_space)(struct sock *sk);
      void (*original_outbound_data_ready)(struct sock *sk);
      void (*original_outbound_error_report)(struct sock *sk);
      void (*original_outbound_destruct)(struct sock *sk);
      ```
        - These pointers store the original socket callbacks for outbound connections. They are used to restore the original behavior when TCP-specific handling is no longer needed.
        - Similar sets of callbacks exist for inbound connections (`original_inbound_*`).

3. **TCP Callback Flags:**
    - ```
      bool tcp_outbound_callbacks_set;
      bool tcp_inbound_callbacks_set;
      ```
        - These flags indicate whether the respective callbacks have been set for the inbound or outbound sockets to avoid pathological corruption of the TCP stack processing.

    .

4. **Connection Timestamps:**
    - `ktime_t outbound_timestamp, inbound_timestamp;`
        - Timestamps that record when the last outbound or inbound connection attempt or handshake exchange was made or successfully established. These timestamps are used to identify which of the potential remote connections to route packets to. Last handshake wins.

.

5. **Socket Address Information:**
   - `struct sockaddr_storage inbound_source, outbound_source, inbound_dest, outbound_dest;`
     - Storage for socket addresses used in inbound and outbound connections, encapsulating both IPv4 and IPv6 address information.

6. **Partial Read Management:**
   - `struct sk_buff *partial_skb;`
     `size_t expected_len;`
     `size_t received_len;`
     - Used to handle partially received data during TCP reads. `partial_skb` stores the current buffer, while `expected_len` and `received_len` track the total expected data length and the amount of data received so far.

7. **Packet Queuing:**
   - `struct sk_buff_head tcp_packet_queue;`
     - Queue for managing TCP packets that are pending transmission or processing.

8. **Work and Scheduling:**
   - `struct delayed_work tcp_retry_work;`
     `struct delayed_work tcp_outbound_remove_work;`
     `struct delayed_work tcp_inbound_remove_work;`
     `struct delayed_work tcp_cleanup_work;`
     - Structures for managing scheduled work items related to TCP connection retries, socket removal, and cleanup.

9. **TCP Connection State Flags:**
   - `bool tcp_retry_scheduled;`
     `bool tcp_outbound_remove_scheduled;`
     `bool tcp_inbound_remove_scheduled;`
     `bool tcp_cleanup_scheduled;`
     - Flags to track whether specific work items are scheduled.

10. **Connection State Flags:**
    - `bool tcp_established;`
      `bool tcp_pending;`
      `bool inbound_connected;`
      `bool outbound_connected;`
        - These flags track the state of the TCP connection:
            1. `tcp_established`: Indicates if a TCP connection has been successfully established.
            2. `tcp_pending`: Indicates if a connection attempt is in progress.
            3. `inbound_connected`: Indicates an established inbound connection.
            4. `outbound_connected`: Indicates an established outbound connection.

11. **Cleanup Flags:**
    - `bool clean_outbound;`
      `bool clean_inbound;`
        - Flags to indicate whether inbound or outbound sockets should be cleaned up at the next opportunity.

12. **Temporary Peer Flag:**
    - `bool temp_peer;`
        - Indicates if the peer is a temporary one, used for handling incoming connections that may not correspond to pre-configured peers.

13. **Send Queue and Lock:**
    - `struct sk_buff_head send_queue;`
      `spinlock_t send_queue_lock;`
        - Queue for managing outgoing packets and a corresponding lock for thread-safe access.

14. **Pending Connection List:**
    - `struct list_head pending_connection_list;`
        - List to manage peers that are awaiting connection handshakes

.

15. **Locks for Synchronization:**
    - ```
      spinlock_t tcp_lock;
      spinlock_t tcp_read_lock;
      spinlock_t tcp_write_lock;
      spinlock_t tcp_transfer_lock;
      ```
        - Spinlocks to protect access to TCP-related states and operations, ensuring thread safety.

16. **Work and Workqueues for Data Processing:**
    - ```
      struct work_struct tcp_read_work;
      struct workqueue_struct *tcp_read_wq;
      struct work_struct tcp_write_work;
      struct workqueue_struct *tcp_write_wq;
      struct work_struct tcp_transfer_work;
      struct workqueue_struct *tcp_transfer_wq;
      ```
        - Work structures and workqueues used to manage asynchronous data processing tasks for reading, writing, and transferring TCP data.

17. **Worker Scheduling Flags:**
    - ```
      bool tcp_read_worker_scheduled;
      bool tcp_write_worker_scheduled;
      bool tcp_transfer_worker_scheduled;
      ```
        - Flags to indicate whether the respective TCP worker (read/write/transfer) is currently scheduled to run.
      ```

18. Delayed and Periodic Worker Routines:

New delayed work items and functions are defined for managing TCP connections, including retries and removal:

```
INIT_DELAYED_WORK(&peer->tcp_retry_work, wg_tcp_retry_worker);
// Work item for TCP connection retries

INIT_DELAYED_WORK(&peer->tcp_inbound_remove_work,
wg_tcp_inbound_remove_worker);  // Work item for inbound TCP
removal

INIT_DELAYED_WORK(&peer->tcp_outbound_remove_work,
wg_tcp_outbound_remove_worker);  // Work item for outbound TCP
removal
```

19. Peer Structure Initialization for TCP:

During peer creation in `wg_peer_create()`, multiple fields and structures are initialized to support TCP functionality. These initializations set up the peer structure to manage TCP connections, handle retries, and manage TCP-specific packet queues:

```
peer->peer_socket = NULL;  // Initialize the peer socket to NULL

skb_queue_head_init(&peer->tcp_packet_queue);  // Initialize the
skb queue for TCP packets

peer->tcp_retry_scheduled = false;  // Initialize TCP retry
scheduled flag

INIT_DELAYED_WORK(&peer->tcp_retry_work, wg_tcp_retry_worker);
// Delayed work for TCP retry

INIT_DELAYED_WORK(&peer->tcp_inbound_remove_work,
wg_tcp_inbound_remove_worker);  // Delayed work for TCP inbound
removal

INIT_DELAYED_WORK(&peer->tcp_outbound_remove_work,
wg_tcp_outbound_remove_worker);  // Delayed work for TCP outbound
removal
```

Modifications to `wg_device` Structure:

Purpose of Modifications:

1. **Support for TCP-Based Connections:** The introduction of TCP listening sockets and listener threads enables WireGuard to establish and manage TCP connections, providing an alternative to UDP, which may be preferable in certain network environments or configurations.
2. **Resource Management and Cleanup:** The cleanup mechanisms (e.g., `tcp_cleanup_work`, related flags, and locks) ensure that the device can efficiently manage resources by cleaning up stale TCP connections. This prevents resource leakage and helps maintain optimal performance.
3. **Thread-Safe Operations:** The use of spinlocks (`tcp_cleanup_lock`, `tcp_connection_list_lock`) ensures that modifications to TCP-related structures are thread-safe, which is crucial in a concurrent environment, especially when multiple threads or cores are involved.
4. **Flexible Endpoint Management:** By keeping track of active connections and their states, the device can dynamically manage its communication endpoints. This includes handling different transport types and dynamically switching between them if needed.

The additional fields in the `wg_device` structure are designed to enable WireGuard to handle TCP connections alongside its traditional UDP-based communication. The structure modifications achieve the following:

1. **TCP Listening Sockets:**
    - **`struct socket __rcu *tcp_listen_socket4, *tcp_listen_socket6;`**
        - These fields store pointers to the TCP listening sockets for IPv4 and IPv6. They are used to accept incoming TCP connections for the WireGuard interface, analogous to the existing UDP sockets (`sock4` and `sock6`).

2. **TCP Listener Threads:**
    - **`struct task_struct *tcp_listener4_thread, *tcp_listener6_thread;`**
        - Threads that handle incoming TCP connections for IPv4 and IPv6. These listener threads continuously wait for incoming connections on the respective TCP sockets and initiate appropriate actions when connections are received.

3. **TCP Connection Management:**
    - `struct list_head tcp_connection_list;`
        - A linked list to keep track of active TCP connections. Each entry in this list corresponds to a TCP connection currently managed by the WireGuard device, facilitating easy traversal and management of connections.

4. **TCP Cleanup Mechanism:**
    - `struct delayed_work tcp_cleanup_work;`
        - A delayed work item that periodically cleans up stale or inactive TCP connections. This helps manage resources efficiently by ensuring that unused connections do not linger indefinitely.
    - `spinlock_t tcp_cleanup_lock;`
        - A spinlock to protect the `tcp_cleanup_scheduled` flag, ensuring that the cleanup process is thread-safe.
    - `bool tcp_cleanup_scheduled;`
        - A flag indicating whether the TCP cleanup work has already been scheduled. This prevents redundant scheduling of cleanup tasks.

5. **TCP Socket Ready Flags:**
    - `bool tcp_socket4_ready;`
      `bool tcp_socket6_ready;`
        - These flags indicate whether the respective IPv4 or IPv6 TCP listening sockets are successfully initialized and ready to accept connections. They prevent repeated initialization attempts once the sockets are set up.

6. **TCP Connection List Lock:**
    - `spinlock_t tcp_connection_list_lock;`
        - A spinlock used to protect access to the `tcp_connection_list`, ensuring that modifications to the list (e.g., adding or removing connections) are thread-safe.

7. **Device Endpoint and Transport:**
    - `struct endpoint device_endpoint;`
        - Stores the device's endpoint information. This could be used for managing source addresses or tracking where connections are being established.
    - `u8 transport;`
        - An indicator of the transport protocol in use.

8. **Listener Activity Status:**
    - `bool listener_active;`
        - A flag indicating whether any of the listener threads (UDP or TCP) are currently active. This helps in managing the state of the listeners and may assist in handling shutdown or restart operations gracefully.