ChatGPT

Emergency Drone Coordination System Projesi İncelemesi

Projenin Amacı

Emergency Drone Coordination System, temel olarak acil durumlarda insansız hava araçlarının (drone) koordineli bir şekilde kazazedelere yardım ulaştırmasını sağlayan bir istemci-sunucu sistemi oluşturmayı hedefler. Projenin amacı, başlangıçta bir **simülasyon ortamından** yola çıkarak bunu **gerçek bir istemci-sunucu mimariye** dönüştürmektir [1]. Bu sistemde drone'lar istemci olarak merkezi bir sunucu ile haberleşerek, rastgele konumlarda ortaya çıkan kazazedelere acil yardım ulaştırmak için koordinasyon sağlar [1]. Bu senaryo, gerçek dünyadaki **drone sürüleri** ile yapılan operasyonlara benzer şekilde tasarlanmıştır (örneğin Zipline'ın medikal teslimat drone'ları veya Amazon'un teslimat drone filoları) [2]. Proje kapsamında eşzamanlı programlama (iş parçacıkları ve mutex'ler), **thread-safe** (iş parçacığı güvenli) veri yapıları ve ağ üzerinden haberleşme gibi becerilerin geliştirilmesi hedeflenmektedir [1].

Kurulum ve Çalıştırma

Proje C programlama diliyle yazılmıştır ve POSIX iş parçacıkları (pthread) kullanılarak geliştirildiği için Linux gibi POSIX uyumlu bir ortamda derlenip çalıştırılması öngörülür. Uygulamanın çalışması için bazı harici kütüphanelerin kurulumu gereklidir. Özellikle görselleştirme için **SDL2** kütüphanesi ve ağ haberleşmesinde kullanılmak üzere **JSON-C** kütüphanesi önerilmektedir [3]. Bu kütüphanelerin sistemde yüklü olduğundan emin olunmalıdır. Proje deposunda bir Makefile bulunması muhtemeldir; bu sayede `make` komutu ile tüm kaynak kod derlenebilir. Aksi takdirde, GCC gibi bir derleyiciyle `*.c` dosyaları derlenirken `-pthread` ve SDL2 için gerekli `-lSDL2` gibi bağlantı seçenekleri kullanılmalıdır.

Derleme tamamlandıktan sonra program çalıştırıldığında bir **SDL penceresi** açılır ve ızgara biçimindeki harita üzerinde drone'lar ile kazazedelerin simülasyonu gerçek zamanlı olarak gösterilir. Uygulama, 10 adet drone'u varsayılan olarak oluşturur ve 40x30 boyutlu bir harita üzerinde başlatır [4]. Program çalışırken drone ve kazazedelerin durumunu sürekli güncelleyerek grafik arayüzde yansıtır. Ekrandaki pencere kapatıldığında (`SDL_QUIT` olayı), program temiz bir şekilde sonlanır ve kullanılan bellek serbest bırakılır [5].

Sistemin Çalışma Prensibi

Proje, **çok iş parçacıklı bir simülasyon** olarak başlayacak şekilde tasarlanmıştır. Uygulama başlatıldığında merkezi sunucu rolünü üstlenen ana proses, öncelikle global veri yapıları ve haritayı oluşturur, ardından çeşitli görevleri paralel yürütmek için alt thread'ler yaratır. Özellikle:

- **Veri Yapılarının Hazırlanması:** Program başlar başlamaz "kazazedeler" (survivors), "yardım edilmiş kazazedeler" (helpedsurvivors) ve "drone'lar" için global listeler oluşturulur [6]. Bu listeler, özel olarak geliştirilen thread-safe `List` veri yapısı kullanılarak sabit kapasiteyle tanımlanır. Ayrıca, harita boyutları (ör. 40 satır x 30 sütun) belirlenip bu boyutlarda bir **grid (ızgara) haritası** oluşturulur [4]. Haritanın her hücresi, içinde o konumdaki kazazedeleri tutacak

kendi **Liste** yapısına sahiptir [7]. Bu hazırlıkların ardından belirli sayıda drone nesnesi yaratılarak her biri için ayrı bir iş parçacığı başlatılır [8].

- **Drone'ların Davranışı:** Her bir drone, uygulama içinde sürekli çalışan bir thread tarafından kontrol edilir. Başlangıçta tüm drone'lar **idle (boşta)** durumdadır ve konumları harita boyutlarına göre rastgele seçilmektedir [9]. Drone thread'leri, sürekli bir döngü içinde kendi durumlarını denetler ve eğer bir göreve (**ON_MISSION**) atanmışlarsa hedef koordinatlarına doğru her seferinde bir adım ilerler [10]. Bu hareket her saniye güncellenir (thread içinde `sleep(1)` ile) ve drone hedefe ulaştığında görevi tamamlayarak tekrar idle durumuna geçer [11]. Örneğin, aşağıdaki kod parçasında görüldüğü gibi bir drone'un x/y konumu hedefe yaklaşacak şekilde her iterasyonda bir birim değiştirilir [10]. Görev tamamlandığında konsola bilgi mesajı basılır (`"Drone X: Mission completed!"`) ve drone yeni görevlere hazır hale gelir. Her drone yapısında, **konum** (`coord`), **hedef** (`target`), **durum** (`status`) ve eşzamanlı erişim için bir **kilit** (`pthread_mutex_t lock`) bulunmaktadır; böylece drone'un durum bilgisi güncellenirken veya okunurken veriler korunur [12] [10].

- **Kazazedelerin Oluşumu:** Sistem çalıştığı sürece ayrı bir iş parçacığı, **kazazede üretici (survivor_generator)** olarak sürekli yeni kazazedeler yaratır. Bu thread bir döngü içinde belirli aralıklarla rastgele bir koordinat seçerek yeni bir *Survivor* nesnesi oluşturur [13]. Her kazazede için benzersiz bir kimlik veya bilgi alanı (`info`) doldurulur ve bulunma zamanı kaydedilir. Oluşturulan kazazede nesnesi, global **survivors** listesine ve ilgili harita hücresinin kendi listesine, kilitli erişim ile eklenir [14]. Böylece, hem genel bir bekleyen kazazedeler kuyruğuna alınmış olur hem de harita üzerinde kendi konumunda kayıt altına alınır. Yeni bir kazazede oluşturulduğunda konsola örneğin `"New survivor at (x,y): SURV-1234"` biçiminde bir mesaj basılır [15]. Kazazedelerin oluşturulma sıklığı rasgeledir; thread her ekleme işleminden sonra 2 ila 5 saniye arası bekler (`sleep`) ve böylece sistemde sürekli ancak kontrollü bir şekilde kazazedeler birikir [15].

- **Görev Atama (AI) Mekanizması:** "**AI controller**" adı verilen ayrı bir iş parçacığı, sürekli olarak bekleyen kazazedelere uygun drone ataması yapmaktan sorumludur. Bu thread, her döngü iterasyonunda öncelikle global **survivors** listesinin başına (en eski eklenmiş, en uzun süredir bekleyen kazazedeye) bakar [16]. Eğer bekleyen en az bir kazazede varsa, sistemdeki **boşta olan drone'lar** arasından ona en yakın olanını belirler. Bu yakınlık basitçe Manhattan mesafesi ($|\Delta x| + |\Delta y|$) olarak hesaplanmıştır [17] [18]. En yakın müsait drone bulunduğunda, bu drone'un durumunu "ON_MISSION" olarak güncelleyip hedef koordinatını ilgili kazazedenin konumu yapar (bu atama `assign_mission()` fonksiyonu ile gerçekleştirilir) [18]. Ardından, ataması yapılan kazazede global bekleme listesinden çıkarılır ve "yardım edildi" olarak işaretlenir [19]. Bu süreç her döngüde tekrarlanarak sistemdeki bekleyen kazazedelerin mümkün olan en kısa sürede bir drone tarafından ele alınması sağlanır. Konsolda, her atama için örneğin `"Drone 3 assigned to survivor at (5, 12)"` ve `"Survivor SURV-1234 being helped by Drone 3"` şeklinde bilgilendirme mesajları gösterilir [18] [20].

- **Harita ve Görselleştirme:** Ana program, yukarıdaki thread'ler arka planda çalışırken, ana iş parçacığında **SDL2 tabanlı bir grafik arayüz**ü yürütür. Harita ızgarası ve içindeki nesneler bu arayüzde sürekli olarak çizilir. `init_sdl_window()` fonksiyonu ile uygun boyutta bir pencere oluşturulur ve her bir hücre için ölçeklendirme ayarlanır (ör. her hücre 20 piksel kare olarak temsil edilir) [21]. Ardından ana döngü, sürekli `draw_map()` çağrısıyla güncel durumu ekrana çizer ve kısa bir gecikme (`SDL_Delay(100)`) ile yenilenir [22]. Çizim sırasında, sistemdeki paylaşımlı verilere güvenli şekilde erişmek için kilit mekanizmaları kullanılır: Örneğin, harita çizilirken her drone'un konumu okunmadan önce o drone'un kendi kilidi kilitlenir ve okuma

bittikten sonra kilit bırakılır [23] [24]. Benzer şekilde, haritanın her hücresindeki kazazede listesine kilitleme ile bakılarak o hücrede kazazede olup olmadığı kontrol edilir [25]. Görselleştirmede bir drone **boştaysa** mavi renkle, **görevdeyse** yeşil renkle gösterilir; kazazedeler ise bulundukları hücrede kırmızı bir kare olarak çizilir [26]. Hatta bir drone görevdeyse, mevcut konumundan hedef kazazedenin bulunduğu konuma doğru **yeşil bir çizgi** çizilerek görevi devam eden drone görsel olarak da vurgulanır [24]. Kullanıcı arayüzü interaktiftir; pencereyi kapatma veya belirli bir tuşa basma gibi olaylar `check_events()` ile izlenir ve kullanıcı pencereyi kapattığında ana döngü sonlanarak program kapanır [27].

Not: Projenin ilerleyen aşamalarında (Phase 2 ve sonrası) sistemin çalışma prensibi gerçek ağ iletişimini de kapsayacak şekilde genişletilecektir. Bu senaryoda her bir drone kendi süreci (programı) olarak çalışacak ve merkezi sunucuya **TCP soket** bağlantısı ile bağlanacaktır [28]. Drone istemcileri periyodik olarak kendi durumlarını (konum, görev durumu vb.) **JSON** formatındaki mesajlarla sunucuya iletecek, örneğin `{ "drone_id": "D1", "status": "idle", "location": [10, 20] }` biçiminde bir JSON mesajı gönderilecektir [29]. Sunucu ise uygun olduğunda drone'lara yeni görev atamasını yine ağ üzerinden JSON mesajlarıyla iletecektir. Bu tam istemci-sunucu modeli, ilk etapta simülasyon olarak gerçekleştirilmiş yukarıdaki mantığın ağ üzerinden dağıtık bir ortamda çalışmasını sağlayacaktır.

Klasör ve Dosya Yapısı

Projenin kod yapısı, anlaşılır şekilde modüllere ayrılmıştır. Ana depo içerisinde **başlık dosyaları** (`.h`) ve **kaynak kod dosyaları** (`.c`) olmak üzere iki temel kısım bulunur. Aşağıda önemli klasör ve dosyaların yapısı özetlenmiştir:

- `headers/` **Klasörü:** Projenin tüm başlık dosyalarını içerir. Bu dosyalar ilgili modüllerin veri yapısı tanımlarını ve fonksiyon bildirimlerini barındırır:
- `globals.h`: Farklı modüller arasında paylaşılan **küresel değişkenlerin** (global listeler, harita vb.) `extern` bildirimlerini yapar. Örneğin, **survivors**, **drones** gibi listelerin ve global **map** nesnesinin burada tanımlandığı varsayılır.
- `list.h`: **List** yapısının tanımını ve liste üzerinde gerçekleştirilen işlemlerin fonksiyon prototiplerini içerir. Özel bir bağlı liste uygulaması olduğundan, `Node` ve `List` yapıları burada tanımlanmıştır. (Not: Bu listeler dizi üzerinde ardışıl bellek kullanan, hem stack hem queue gibi davranabilen bir yapıdadır.)
- `map.h`: **Harita** yapısını tanımlar. Harita (Map) yapısı, grid'in boyutlarını (`width`, `height`) ve hücrelerin 2 boyutlu dizisini tutar. Ayrıca her bir **MapCell** yapısı içinde o hücreye ait koordinat ve kazazede listesi bulunur. `map.h` içinde global `Map map` değişkeni `extern` ile bildirilir.
- `drone.h`: **Drone** yapısını tanımlar. Her bir drone için kullanılacak özellikler (ör. `id`, konum (`coord`), hedef (`target`), durum (`status` – ör. IDLE veya ON_MISSION), thread id (`pthread_t`) ve kilit (`pthread_mutex_t`) gibi) bu yapıda tanımlanır. Ayrıca drone'larla ilgili fonksiyonların (initialize, davranış vs.) bildirimleri bu dosyadadır.
- `survivor.h`: **Survivor** (kazazede) yapısını tanımlar. Bir kazazedenin konumu (`Coord`), keşfedilme zamanı (`discovery_time`), durum bilgisi ve opsiyonel olarak yardım alındıysa bunun zamanını ve kimliğini içeren alanlar bu yapıda tanımlanır. Yeni kazazede oluşturma (`create_survivor`) ve kazazedeyi temizleme (`survivor_cleanup`) gibi fonksiyonlar burada prototiplenir.
- `ai.h`: **Yapay zekâ kontrol modülü** için bildirimler içerir. Drone atama algoritmasına dair fonksiyonların (ör. `ai_controller`, `find_closest_idle_drone`, `assign_mission` gibi) prototipleri bu dosyada tanımlıdır.

- `view.h`: **Görselleştirme** ile ilgili fonksiyon bildirimlerini içermesi amaçlanan dosyadır. SDL tabanlı çizim fonksiyonları (`init_sdl_window`, `draw_map`, `draw_drones`, `draw_survivors`, `check_events`, `quit_all` vb.) burada prototiplenmiş olabilir. (*Not:* Kod içinde `view.h` include edilmeden fonksiyonların tanımlandığı görülüyor, bu nedenle bu başlık dosyası minimal içerikte veya opsiyonel olabilir.)

- **Kaynak Kod Dosyaları (`*.c`):** Projenin asıl işleyişini gerçekleştiren C dosyalarıdır. Her biri yukarıdaki başlık dosyalarına karşılık gelen implementasyonlar içerir:

- `controller.c`: **Ana kontrol ve giriş noktası**. `main()` fonksiyonunu içerir [30]. Programın başlangıcında tüm listeleri ve haritayı oluşturur, diğer modüllerin başlangıç fonksiyonlarını çağırarak drone, kazazede ve AI thread'lerini başlatır [31]. SDL penceresini açar ve ana çizim döngüsünü yürütür [22]. Program sonlandığında bellek temizleme ve kaynakları serbest bırakma işlemlerini gerçekleştirir [32].
- `list.c`: **List veri yapısının implementasyonu**. `list.h`'de tanımlanan List yapısının fonksiyonlarını burada buluruz. Ekleme (`add`), silme (`remove` / `removedata`), yığıt mantığında çekme (`pop`), gözleme (`peek`) ve listeyi yok etme (`destroy`) gibi fonksiyonlar tanımlıdır. Bu implementasyon, listede kullanılan düğümleri tek bir blok bellek içinde tutarak verimli bellek yönetimi sağlar. İş parçacığı güvenliği için `List` yapısına bir `pthread_mutex_t` kilidi eklenmiş ve her işlem bu kilit ile korunmuştur (ör. `list->add` çağrıları kilitlenerek yapılır). Kod üzerinde eğitim amaçlı bırakılmış bazı **TODO** notları vardır (ör. senkronizasyon ekleme, `destroy` fonksiyonunda gereksiz `memset`'in kaldırılması gibi) [33] [34].
- `map.c`: **Harita modülünün implementasyonu**. Haritanın başlatılması (`init_map`) ve temizlenmesi (`freemap`) fonksiyonları bu dosyada yer alır. `init_map`, belirli boyutlarda (parametre olarak verilen yükseklik ve genişlik) bir ızgara oluşturur, her hücre için bellek ayırır ve o hücreye ait boş bir kazazede listesi oluşturur [7]. `freemap` ise program sonlanırken her hücrenin listesini yok eder ve ayırılmış bellek alanlarını serbest bırakır [35].
- `drone.c`: **Drone modülünün implementasyonu**. Tüm drone'ları başlatan `initialize_drones` fonksiyonunu ve her bir drone'un ayrı thread'de çalıştırdığı `drone_behavior` fonksiyonunu içerir. `initialize_drones`, belirli bir sayıda drone için bellek ayırır, her birine rastgele başlangıç koordinatı atar ve durumlarını IDLE olarak işaretler [9]. Her drone için bir `pthread_mutex_init` ile kilit oluşturulur ve global **drones** listesine eklenir (kilitlenerek) [12]. Ardından `pthread_create` ile `drone_behavior` fonksiyonu yeni bir thread'de yürütülmeye başlanır [36]. `drone_behavior` içinde, drone'un durumuna göre hareket ettirilmesi ve görev tamamlanma kontrolü yapılır [10] [11]. Bu dosya ayrıca `cleanup_drones` gibi fonksiyonlarla drone thread'lerinin iptalini ve kilitlerin yok edilmesini de ele alır [37].
- `survivor.c`: **Kazazede (Survivor) modülünün implementasyonu**. Yeni bir kazazede yapısı oluşturan `create_survivor` fonksiyonunu ve sürekli kazazede üreten `survivor_generator` thread fonksiyonunu içerir. `survivor_generator` fonksiyonu, rastgele konum ve kimlik bilgisi ile periyodik olarak (2-5 saniyede bir) yeni bir Survivor nesnesi yaratır [13] [15]. Oluşturulan bu nesneyi thread-safe şekilde global **survivors** listesine ve ilgili harita hücresinin kendi listesine ekler [14]. Üretilen kazazedeler konsola loglanır ve sonsuz döngü şeklinde program sonuna dek devam eder. Dosyada ayrıca bir kazazedeyi temizlemek için `survivor_cleanup` fonksiyonu tanımlıdır (harita hücresindeki listeden çıkarıp belleğini serbest bırakmak için) [38].
- `ai.c`: **Yapay zekâ kontrol modülünün implementasyonu**. Bu dosya, kazazedeleri drone'lara atayan mantığı gerçekleştirir. En önemli fonksiyonu `ai_controller` adlı thread fonksiyonudur. `ai_controller`, global kazazede listesinde bekleyen ilk elemanı alır, tüm drone'lar arasında uygun (boşta ve en yakın) bir drone'u bulur ve o drone'a görevi atar [16]. Atama işlemi

`assign_mission` yardımıyla drone'un iç durumunu güncelleyerek yapılır [39] [18]. Sonrasında ilgili kazazedeyi bekleme listesinden çıkarıp durumunu günceller [19]. Ayrıca `find_closest_idle_drone` adıyla, verilen bir hedef koordinata göre drone listesindeki en yakın boşta drone'u bulan yardımcı bir fonksiyon mevcuttur [40]. Kod içinde olası deadlock durumlarını önlemek ve yardım edilen kazazedeyi ayrı bir listeye almak gibi konularda **TODO** yorumları bırakılmıştır [41] [42].

- `view.c`: **Görselleştirme modülünün implementasyonu**. SDL2 kütüphanesini kullanarak grafik arayüzü oluşturan ve sistemin durumunu 2D olarak çizen fonksiyonları içerir. `init_sdl_window()` fonksiyonu, harita boyutlarına göre bir pencere ve renderer oluşturur [21]. `draw_map()` fonksiyonu, haritanın her hücresi için hücrede bir kazazede varsa kırmızı renkli bir kare çizmek, her bir drone'un konumuna göre mavi/yeşil bir piksel çizmek ve görevdeki drone'lar için hedefe uzanan bir çizgi çizmek gibi işlemleri yapar. Bu çizim işlemleri sırasında `drone_fleet` dizisindeki her drone verisine ve `map.cells` içindeki her **survivors** listesine mutex kilidi ile güvenli erişim sağlanır (örneğin `pthread_mutex_lock(&drone_fleet[i].lock)` ile) [23] [24]. Kullanıcı etkileşimleri `check_events()` ile izlenir; pencere kapatma olayı yakalandığında fonksiyon `1` döndürerek ana döngüyü kırar ve `quit_all()` çağrısıyla SDL kaynakları temizlenir [27] [43].
- **Doküman Dosyaları:** Depoda kod dışında, projenin aşamalarını ve protokol detaylarını anlatan Markdown dokümanları da bulunur. Örneğin, `README.md` projenin genel tanımı ve üç aşamalı geliştirme planını içerirken, **Lab for Phase-1.md** dosyası birinci fazın laboratuvar talimatlarını ayrıntılı olarak açıklamaktadır. Ayrıca README içinde atıf yapılan **communication-protocol.md** dosyası, istemci-sunucu mesajlaşma formatını tanımlamak üzere tasarlanmıştır; ancak bu dosya repoda mevcut değildir (muhtemelen henüz oluşturulmamış veya tamamlanmamıştır).

Kodun Genel İşleyişi ve Ana Bileşenler

Kodun yapısı incelendiğinde, sistemin birbirleriyle etkileşimli birkaç ana bileşen etrafında inşa edildiği görülür. Genel işleyiş, **üretici-tüketici** benzeri bir model ile **paylaşılmış veri yapıları** üzerinden koordine edilir. Başlıca bileşenler ve işleyişleri şöyle özetlenebilir:

- **İş Parçacıkları ve Eşzamanlılık:** Uygulama aynı anda çalışan birden fazla thread barındırır: Drone'lar, kazazede üreteci ve AI kontrolcüsü ayrı iş parçacıklarında paralel yürütülürken ana iş parçacığı grafik arayüz döngüsünü yönetir [44]. Bu çoklu iş parçacığı yapısı nedeniyle, ortak kullanılan verilere erişimde **eşzamanlılık denetimi** kritik önemdedir. Kod içerisinde her paylaşılan veri yapısı bir **mutex kilidi** ile korunmuştur. Örneğin, global **survivors** listesine yeni bir kazazede eklenirken önce listenin kilidi alınır, ekleme yapılıp bittiğinde kilit geri bırakılır [14]. Benzer şekilde, drone'ların global listesinde arama yapılırken veya harita hücresindeki listeye erişilirken ilgili liste kilitlenir [17]. Bu sayede, bir thread listenin üzerinde işlem yaparken bir başka thread'in aynı listeye erişimi engellenerek **veri yarışı (race condition)** durumları önlenir. Drone nesnelerinin kendi iç durumları (konum, hedef, durum bayrakları) da her bir drone için tanımlanmış ayrı mutex'lerle korunmaktadır; çizim veya görev atama sırasında drone'un verileri güncellenirken bu kilitler kullanılmaktadır [23] [24].

- **Thread-Safe Liste Yapısı:** Projenin kalbinde, birden fazla thread tarafından güvenli şekilde kullanılabilen özel bir **Liste** veri yapısı bulunmaktadır. Bu `List` yapısı, hem **kazazedelerin kuyrukları** (bekleyen ve yardım edilen) hem de **drone kayıtları** için ortak olarak kullanılır. Liste implementasyonu, bellekte ardışıl bir alan ayırarak düğümleri burada saklayan, böylece ekleme/çıkarma işlemlerini O(1) bellek ayrımı ile yapabilen bir tasarımdadır. Her `List` yapısında bir `pthread_mutex_t` **kilit** alanı tanımlanmış ve tüm temel işlemler (ekleme, silme, okuma vb.) bu kilit ile sarılmıştır [45]. Örneğin, `survivors->add()` çağrıları yapılmadan önce `survivors-`

`>lock` kilitlenir, işlem tamamlandıktan sonra kilit açılır [14]. Bu liste yapısı, **düğümler arası çift bağlantılar** (prev-next işaretçileri) ile hem baştan hem sondan gezinmeye izin verirken, bir yandan da tüm düğümler tek bir blok içinde tutulduğu için bellek erişim örüntüsü bakımından verimlidir. Liste üzerinde arama ve çıkarma işlemleri için gereken yardımcı fonksiyonlar (`find_memcell_fornode`, `removenode` vs.) `list.c` içinde tanımlanmış ve `List` yapısı içinde fonksiyon işaretçileri olarak saklanmıştır [46]. Sonuç olarak, **survivors**, **helpedsurvivors** ve **drones** gibi global listeler ile harita hücrelerindeki yerel listeler aynı altyapıyı kullanarak tutarlı bir arayüzle yönetilir [47].

- **Drone Filosu Yönetimi:** Drone'lar sistemin hareketli ajanlarıdır. Kod içerisinde `Drone` yapısı, her bir drone'un ihtiyaç duyduğu tüm bilgileri kapsar ve global bir dizi (`drone_fleet`) halinde tutulur [48]. Program başlatıldığında `initialize_drones()` fonksiyonu her drone için bellek ayırır, gerekli başlangıç değerlerini atar (rastgele konum, IDLE durum vb.) ve her birini global **drones** listesine ekler [9] [49]. Bu tasarım, hem dizide indeksleme ile doğrudan erişime izin vermekte hem de tüm drone nesnelerinin thread-safe listede de referans olarak bulunmasını sağlamaktadır. Drone davranışı, `drone_behavior` fonksiyonu ile modellenir: Bu fonksiyon her drone için ayrı bir thread'de sürekli çalışır ve drone'u hedeflerine doğru ilerletir [10]. Her adımda drone'un kilidi kilitlenerek konum değişikliği atomik bir şekilde yapılır ve durum kontrolü gerçekleştirilir. Böylece bir drone uçarken konumu hem kendi thread'i tarafından güncellenir hem de diğer thread'ler (AI veya view) tarafından güvenli biçimde okunabilir. Drone'ların global listede tutulması, AI modülünün kolaylıkla tüm drone'ları tarayarak uygun olanı seçebilmesine imkân tanır [17]. Ayrıca gerekirse yeni bir drone eklemek veya bir drone'u listeden çıkarmak (ör. bağlantısı kesilen bir drone istemcisi olduğunda) bu yapı üzerinden senkronize şekilde yapılabilecektir.

- **Kazazedelerin Takibi:** Sistemde kazazedeler dinamik olarak üretildiği için bunların etkin yönetimi önemlidir. Her yeni kazazede, **survivors** adlı global listede toplanır ve bu liste bir **kuyruk** gibi davranır (ilk giren kazazede en önce değerlendirilir) [16]. Ayrıca harita yapısı sayesinde her kazazede kendi konum hücresinin listesine de eklenir; bu sayede görselleştirme modülü doğrudan doğruya ilgili hücrede bir kazazede olup olmadığını hızlıca kontrol edebilir. **AI modülü**, survivors listesinin en başındaki öğeyi (en uzun süredir bekleyen kazazedeyi) alıp işlemeye çalışır [16]. Ataması yapılan kazazedeler, mevcut implementasyonda survivors listesinden çıkarılıp durumları güncelleniyor, fakat henüz sistemde "yardım edilmiş kazazedeler" listesinin tam kullanımı implementasyon aşamasındadır (bu konu eksik kısımlarda ele alınmıştır). İleride, yardım edilmiş kazazedelerin **helpedsurvivors** listesine taşınarak ayrı izlenmesi ve belki istatistiksel hesaplamalarda kullanılması hedeflenmektedir. Kazazedelerin verileri (isim/id alanı, bulunduğu zaman, vs.) Survivor yapısında sabit olarak tutulur ve bellek yönetimi açısından bir kazazede işinin bitmesiyle `free` edilerek temizlenmesi gerekir – mevcut kodda bu işlem henüz tam entegre edilmemiştir ve bir TODO olarak işaretlenmiştir [42].

- **Yapay Zekâ ve Görev Atama: AI (yapay zekâ) bileşeni**, sistemdeki koordinasyonun beynidir. Amaç, adil ve verimli bir şekilde her kazazedeye bir drone'un yönlendirilmesidir. AI modülü basit bir strateji uygular: En uzun süredir bekleyen kazazedeyi al ve ona en yakın boşta drone'u ata [18]. Bu yaklaşım, bir yandan kazazedelerin mümkün olduğunca bekleme süresini azaltmaya çalışırken diğer yandan da her drone'un konumunu dikkate alarak toplam seyahat sürelerini düşürür. Kodda `find_closest_idle_drone()` fonksiyonu bu işi yapar; drone listesi üzerinde dolaşarak IDLE durumunda olanlar içinde hedefe Manhattan mesafesi en küçük olanı bulur [17]. Uygun drone bulunduğunda `assign_mission()` ile drone'un hedef koordinatı ilgili kazazedenin konumu olarak ayarlanır ve drone'un durumu ON_MISSION yapılır [39] [18]. Bu atama sırasında hem drone'un kendi kilidi hem de listelerin kilidi doğru sırada kullanılarak thread'ler arası tutarlılık sağlanır. AI thread'i bu işlemleri yaparken **survivors listesini**

kilitleyerek kazazedeye erişir ve atama tamamlanınca kilidi bırakır [41] [50]. Mevcut durumda bu kilitleme işlemlerinin olası deadlock yaratmaması için dikkat edilmesi gereken noktalar kod içerisinde yorumlarla belirtilmiştir (örneğin önce kazazedeler listesi, sonra drone listesi kilitleniyor; tersi durumda risk olabileceğine dair not düşülmüş) [41]. İleride AI algoritması geliştirilmeye açıktır; örneğin kazazedelere öncelik derecesi atamak (kritik yaralılar öncelikli gibi) veya drone'lar arasında görev dağılımını dengelemek (her drone'a eşit iş yükü vermek) gibi iyileştirmeler eklenebilir.

- **Grafik Arayüz ve İzleme: SDL2 tabanlı grafik arayüz**, sistemin durumunu gerçek zamanlı izlemeyi sağlar. `view.c` modülü, global **map** yapısını kullanarak her döngüde haritayı yeniden çizer. Bunun için `draw_map()` fonksiyonu, haritadaki her hücre için `draw_survivors()` ve tüm drone'lar için `draw_drones()` yardımcı fonksiyonlarını çağırır. `draw_drones()` içinde drone filosu dizisi dolaşılarak her drone'un durumu kontrol edilir; drone boşta ise mavi, görevde ise yeşil renkte bir piksel ilgili konuma çizilir [23]. Eğer görevdeyse ayrıca drone'un mevcut konumu ile hedefi arasına yeşil bir çizgi çizilerek görev rotası görselleştirilir [24]. Bu süreçte her drone'un verisi `drone_fleet[i].lock` ile kilitlenerek okunur ve çizim sonrası kilit açılır, böylece hareket halindeki drone'un verileriyle çizim arasında uyumsuzluk olmaz [23] [24]. Benzer şekilde `draw_survivors()` fonksiyonu harita hücrelerindeki survivor listelerini kontrol eder; eğer bir hücre listesinde en az bir kazazede varsa o hücreye kırmızı bir kare çizilir [25]. Bu işlem de her hücrenin `survivors->lock` kilidi alınarak yapılır ve hemen ardından bırakılır [25]. Kullanıcı arayüzü, SDL olaylarını sürekli poll ederek pencere kapatma veya klavye girişlerini denetler [27]. Kullanıcı pencereyi kapattığında `check_events()` ana döngüye çıkması için sinyal verir ve programın kapanma süreci başlar [27]. Görselleştirme modülü sayesinde sistemin eşzamanlı işlemleri (drone hareketleri, yeni kazazedeler, görev atamaları) anlık olarak takip edilebilir ve demo amacıyla sunulabilir. İleride gerçek istemci-sunucu sürümde, bu görselleştirme ayrı bir araca dönüştürülebilir (örneğin web tabanlı bir arayüz veya ayrı bir monitor programı).

Kullanılan Teknolojiler

Proje geliştirilirken aşağıdaki teknoloji ve araçlar kullanılmaktadır:

- **Programlama Dili:** C dilinde yazılmıştır. Sistem seviyesinde bellek yönetimi ve iş parçacığı kontrolü için C dilinin esnekliği ve performansından yararlanılmıştır. Kod POSIX standartlarına uygun olup <pthread.h> gibi kütüphaneler içermektedir [51].

- **İşletim ve Eşzamanlılık: POSIX Threads (pthread)** kütüphanesi kullanılarak çoklu iş parçacığı uygulaması gerçekleştirilmiştir. Mutex kilitleri ve thread oluşturma/iptal fonksiyonları doğrudan pthread arayüzü ile yapılır. Örneğin, her liste yapısına `pthread_mutex_t` eklenmesi ve kilitlenmesi, pthread'in mutual exclusion mekanizması ile sağlanır [45].

- **Veri Yapıları: Özel Liste (List) Yapısı**, C dilinde düşük seviyeli bellek yönetimi teknikleriyle (malloc, pointer aritmetiği) uygulanmıştır. Bu veri yapısı, klasik bağlı liste mantığını tek bir blok bellek üzerinde gerçekleştirerek, düğümler arası gezinmede ve bellek ayırımında verimlilik sağlamaktadır. Ayrıca, C dilinin fonksiyon pointer'ları kullanılarak nesne yönelimli bir tarzda tasarlanmıştır (List nesnesi kendi işlemlerini fonksiyon pointer'ları olarak bünyesinde taşır).

- **Ağ İletişimi:** Planlanan istemci-sunucu mimarisi için **TCP/IP soket** programlama kullanılacaktır. Phase-2 itibarıyla her drone istemcisi bir soket üzerinden sunucuya bağlanacak, sunucu ise çoklu istemciyi yönetmek için birden fazla thread kullanacaktır [52] [53]. Şu anki kod tabanında soket

programlama henüz entegre edilmemiş olsa da, ileride `socket()`, `bind()`, `accept()`, `connect()` vb. sistem çağrılarıyla klasik BSD soket yapısı kullanılacaktır.

- **Veri Formatı:** İstemci-sunucu arasındaki mesajlaşma için **JSON** formatı tercih edilmiştir. Drone'ların durum güncellemeleri ve sunucunun görev atama mesajları insan tarafından okunabilir basit JSON nesneleri olarak tanımlanmıştır [29] [54]. Bu JSON verilerinin oluşturulup ayrıştırılması için C dilinde popüler bir kütüphane olan **JSON-C** önerilmektedir [55]. JSON-C kütüphanesi, C yapıları ile JSON metinlerini kolayca dönüştürmeye imkân verecektir (örn. drone durumunu JSON string'e çevirip göndermek, gelen JSON komutlarını parse etmek).

- **Grafik Arayüz: SDL2 (Simple DirectMedia Layer)** kütüphanesi kullanılmaktadır [55]. SDL2, C ile düşük seviyede grafik, pencere ve olay yönetimini kolaylaştıran bir multimedyal kütüphanedir. Bu proje özelinde SDL2 sadece 2B grafikler ve temel olay yönetimi için kullanılır (OpenGL gibi ileri grafik özelliklerine ihtiyaç yoktur). SDL ile pencere açma, renderer oluşturma, primitif şekiller çizme (dikdörtgen, çizgi, nokta) ve klavye/pencere olaylarını yakalama işlemleri gerçekleştirilir [21] [23]. Platform bağımsız yapısı sayesinde SDL2 kullanımı projenin farklı sistemlerde (Linux, Windows, macOS) derlenip çalıştırılmasını da kolaylaştırır.

- **Geliştirme Ortamı:** Proje genel olarak bir eğitim projesi olduğundan, muhtemelen Unix tabanlı bir ortamda (ör. bir Linux dağıtımı) geliştirilmiştir. Derleme için **GCC** veya benzeri bir C derleyicisi kullanıldığı varsayılabilir. Kontrol sistemi olarak Git ve GitHub kullanılmıştır (proje deposu GitHub üzerinde tutulmaktadır). Kod üzerinde açıklama yorumları ve README dokümanları, projenin öğretici yönünü vurgular niteliktedir. Ayrıca, olası bellek hatalarını yakalamak için `valgrind` gibi araçlar veya eşzamanlılık sorunlarını tespit için `helgrind` gibi araçlar geliştirme sürecinde kullanılabilir.

Eksik veya Geliştirilmeye Açık Kısımlar

Proje genel olarak bir prototip ve eğitim uygulaması olarak çalışır durumdadır; ancak inceleme sonucunda bazı kısımlarının eksik bırakıldığı veya geliştirilebileceği görülmüştür:

- **Ağ (Socket) İletişiminin Eksikliği:** Projenin ikinci aşaması olarak planlanan **gerçek istemci-sunucu haberleşmesi** henüz kod tabanında uygulanmamıştır. README dokümanında Phase-2 için "simüle edilen thread'leri gerçek soket tabanlı istemci ve sunucularla değiştirme" hedefi belirtilmesine rağmen [28], mevcut depodaki uygulama hala tek bir proses içinde çalışmaktadır. Yani drone'lar ayrı birer istemci programı olarak değil, ana programın thread'leri olarak çalışıyor. TCP bağlantıları, veri alışverişi (JSON mesajları) ve çoklu proses senkronizasyonu gibi konular gerçekleştirilmemiş durumdadır. Bu, projenin hedeflenen nihai mimarisi için önemli bir eksiktir ve bir sonraki geliştirme adımında ele alınmalıdır.

- **"HelpedSurvivors" Listesinin Kullanılmaması:** Kodda global olarak yaratılan **helpedsurvivors** adlı bir liste bulunmaktadır [56]. Bu liste, yardım edilmiş (drone ulaşmış) kazazedeleri takip etmek amacıyla düşünülmüştür. Ancak mevcut implementasyonda bir kazazede bir drone'a atandığında survivors listesinden çıkarılmakta fakat helpedsurvivors listesine eklenmemektedir. `ai.c` içinde bu işlemi yapmaya yönelik kod bölümü yorum satırına alınmış (devre dışı bırakılmış) ve bir "TODO" notu düşülmüştür [42]. Dolayısıyla, **helpedsurvivors listesi şu an kullanılmıyor** ve kazazedelerin görev tamamlandığında takibi eksik kalıyor. Bu durum, kazazedelerin bellekten temizlenmesi veya istatistiksel bilgilerin toplanması açısından bir eksiklik yaratabilir. İleride bu listenin doğru şekilde kullanılması (örn. bir drone hedefe ulaştığında kazazedeyi bu listeye taşıma) gerekecektir.

- **Kazazedelerin Haritadan Tamamen Kaldırılması:** Bir drone görevini tamamladığında, ilgili kazazedeye ulaştığı varsayılır. Mevcut uygulamada kazazede, global listeden çıkarılıyor ancak harita üzerindeki hücresel listeden çıkarılması hemen yapılmıyor. `survivor_cleanup` fonksiyonu tek bir kazazedeyi harita hücresinden silip bellekten atmak için tanımlı olsa da [38], bu fonksiyonun ne zaman çağrılacağı net değildir. `ai.c` içerisinde, kazazedenin harita hücresindeki listeden silinmesiyle ilgili kod da yorum halinde bırakılmıştır [57]. Bu da, halihazırda harita üzerinde kurtarılan kazazedelerin kırmızı kare olarak görünmeye devam edebileceği (ya da bellek sızıntısına yol açabileceği) anlamına gelir. Bu bölümün tamamlanması, örneğin drone hedefe ulaştığında ilgili kazazedeyi harita listesinden çıkarmak ve bellek temizliğini yapmak şeklinde olmalıdır.

- **Deadlock (Kilitlenme) Riski:** Eşzamanlı sistemlerde doğru kilit sıralaması önemlidir. Kod incelendiğinde, AI thread'inin önce **survivors listesinin**, sonra içinde çağırdığı `find_closest_idle_drone` ile **drones listesinin** kilidini aldığı görülür [17] [16]. Başka bir parça ise drone ekleme sırasında (initialize_drones) **drones listesini** kilitlemektedir [49]. Şu anki kullanımda belirgin bir deadlock durumu yaşanmıyor gibi görünse de, kodda geliştiricinin kilit sırası konusunda bir uyarı bıraktığı ve kilitleme/kilit açma yerlerinin olası deadlock'ları önlemek için değiştirilmesi gerektiğini belirttiği görülmüştür [41]. Özellikle ileride daha karmaşık senaryolarda (örn. drone disconnect olayı aynı anda kazazede ataması ile çakışırsa) kilitlenmeler yaşanabileceği için bu konuya dikkat edilmelidir. Mevcut kodda bu risk henüz proaktif bir çözüm ile giderilmemiştir (TODO olarak durmaktadır).

- **Hafıza Yönetimi ve Hatalar:** Kodda işaret edilen ufak bazı bellek yönetimi düzeltmeleri henüz yapılmamıştır. Örneğin, `List->destroy` fonksiyonunda önce liste verisinin belleği `free` edilip sonra `memset` ile listenin sıfırlanması gibi gereksiz bir adım vardır [34]. README'de bu redundant `memset` çağrısının kaldırılması gerektiği not edilmişti, ancak kodda halen duruyor. Bu durum ciddi bir bellek sızıntısı olmasa da gereksiz bir işlemdir. Aynı şekilde, `listtest.c` gibi test amaçlı bir dosyada `sprintf` yerine `snprintf` kullanılması önerilmişti [45], fakat depo içinde böyle bir test dosyası bulunmadığı için bu değişiklik muhtemelen uygulanmamıştır. Ayrıca, drone'ların ve diğer thread'lerin sonlandırılması konusunda da iyileştirme alanı vardır: Şu an program kapatıldığında drone thread'leri `pthread_cancel` ile iptal edilip kilitleri yok ediliyor [37], fakat **survivor_generator** ve **ai_controller** thread'leri için benzer bir temizlik mekanizması yok (program sonlanırken bu thread'ler aktif ise otomatik sonlanıyor). Daha kontrollü bir kapanış için bu thread'lerin de iptal edilip join edilmesi düşünülebilir.

- **Planlanan İyileştirmelerin Eksikliği:** Projenin Phase-3 olarak tanımlanan ileri aşamasında bir dizi iyileştirme ve ek özellik planlanmıştır – örneğin **yük testleri (50+ drone ile)**, **hata toleransı** (bağlantısı kopan drone'ların tespiti ve görevlerin yeniden atanması), **önceliklendirme (QoS)** gibi konular [58]. Mevcut kod tabanında bu konular henüz ele alınmamıştır. Özellikle bir **heartbeat (kalp atışı sinyali)** mekanizması veya drone'ların belirli süre mesaj yollamazsa düşmesi senaryosu implemente edilmemiştir. Aynı şekilde, bir **performans izleme** fonksiyonu (ör. ortalama kazazede bekleme süresi veya drone kullanım oranı hesaplama) belirtilmesine rağmen kodda yer almıyor [59]. **Web tabanlı gösterge paneli** veya sunucudan ayrı bir müşteri tarafı arayüzü de opsiyonel öneri olarak sunulmuş ancak gerçekleştirilmemiştir [60]. Bu özelliklerin yokluğu, projenin eğitsel hedefleri kapsamında normal olsa da, tam bir ürün uygulaması için geliştirilmesi gereken alanlardır.

Geliştirme İçin Yol Haritası

Mevcut durum ve yukarıda belirtilen eksikler göz önüne alındığında, **Emergency Drone Coordination System** projesini ilerletmek veya kullanıma hazır hale getirmek isteyenler için aşağıdaki adımları içeren mantıklı bir yol haritası önerilebilir:

1. **Mevcut Simülasyonun Stabilize Edilmesi:** İlk adım olarak, **Phase-1** kapsamında gerçekleştirilen mevcut çok iş parçacıklı simülasyon eksiksiz ve hatasız hale getirilmelidir. Bu aşamada:
2. Kod içindeki **TODO** ile belirtilen sorunlar giderilmelidir. Örneğin, `List->destroy` fonksiyonundaki gereksiz işlemler kaldırılarak hafıza yönetimi temizlenmeli [34], ve kilit sıralaması potansiyel deadlock ihtimalini tamamen ortadan kaldıracak şekilde düzenlenmelidir [41].
3. **helpedsurvivors** listesinin kullanımı hayata geçirilmelidir. Bir drone hedefe ulaştığında ilgili kazazedeyi bu listeye ekleyip, hem global survivors listesinden hem de harita hücresinden çıkarmak için gereken kod yazılmalıdır [42] [57]. Bu, kazazedelerin bellekten temizlenmesi ve ileride istatistiksel analiz yapılabilmesi için temel oluşturur.
4. Simülasyon senaryosu farklı koşullarda test edilmelidir: Örneğin drone sayısını artırıp azaltarak (10 yerine 5 veya 20 drone ile) sistem davranışı gözlemlenmeli, beklenmeyen yarış koşulları veya kilitlenmeler olup olmadığı kontrol edilmelidir. Gerekirse **Helgrind** gibi araçlarla çoklu thread senkronizasyonu doğrulanmalıdır.

5. Bu aşamada, kodun anlaşılabilirliğini artırmak için yorumlar güncellenebilir ve README dokümanı, yapılan düzeltmeleri yansıtacak şekilde revize edilebilir.

6. **İstemci-Sunucu Mimarisi ve Ağ İletişiminin Gerçeklenmesi:** İkinci adım, projenin esas hedefi olan gerçek istemci-sunucu modeline geçiş olacaktır. Bu kapsamda:

7. **Sunucu Uygulaması:** Mevcut "controller" kodu bir sunucu programına dönüştürülmelidir. Sunucu, başlangıçta sadece merkezi bileşenleri (harita, global listeler, AI ve görselleştirme) başlatmalı, ardından bir **TCP sunucu soketi** oluşturup dinlemeye başlamalıdır. Her gelen drone bağlantısı için ayrı bir thread oluşturularak (ör. `handle_drone` fonksiyonu) o bağlantı yönetilmelidir [61]. Sunucu, bağlı drone'ların durumlarını bir listede tutmalı ve periyodik olarak onlardan gelen verileri okuyup gerekli atamaları yapmalıdır. README'de verilen örnek sunucu döngüsü (accept ile yeni bağlantı ve thread yaratma) bu noktada rehber olacaktır [53].
8. **Drone İstemci Uygulaması:** Her bir drone artık ayrı bir program (veya işlem) olacağından, yeni bir **drone client** uygulaması geliştirilmelidir. Bu uygulama, belirlenen sunucu adresine bağlanarak kendini kaydedecek, ve belli aralıklarla sunucuya durumunu bildirecektir [62]. Mevcut simülasyon kodundan `drone_behavior` fonksiyonu büyük ölçüde yeniden kullanılabilir; ancak bu sefer konum güncellemelerinin yanı sıra sunucuya **STATUS_UPDATE** mesajları göndermesi gerekecek. Örneğin, drone istemcisi her 1 saniyede bir kendi `id`, `status` ve `location` bilgisini JSON formatında sunucuya iletebilir [62]. Sunucudan bir görev atama mesajı aldığında (örneğin `{"type": "mission", "target": [x, y]}` gibi), bunu ayrıştırıp kendi hedef koordinatını güncelleyerek `drone_behavior` döngüsüne entegre etmelidir.
9. **İletişim Protokolü:** İstemci ve sunucu arasında net bir protokol tanımlanmalıdır. README'de öngörülen protokolde, drone'ların **STATUS_UPDATE** ve **MISSION_COMPLETE** gibi mesajlar yollayacağı, sunucunun ise **ASSIGN_MISSION** ve periyodik yoklama (**HEARTBEAT**) mesajları göndereceği belirtilmiştir [54]. Bu mesajlar için **JSON şemaları** belirlenmeli ve her iki tarafta da JSON verilerini işleyecek kod yazılmalıdır. Bu amaçla **JSON-C kütüphanesi** kullanılabilir – örneğin sunucu tarafında `json_tokener_parse()` ile gelen string parse edilip komut nesnesine dönüştürülebilir, istemci tarafında bir durum struct'ı `json_object` API ile stringe çevrilebilir.
10. **Senkronizasyon ve Yapı:** Sunucu, çoklu istemciden gelen mesajları yöneteceği için, mevcut thread-safe listeler yapısı burada da işe yarayacaktır. Örneğin sunucuda global **drones** listesi

zaten var; ancak artık bu listede gerçek soket bağlantısı olan drone'lar bulunacak. Sunucu thread'leri (her drone bağlantısı için bir tane) gelen mesajı alır almaz ilgili drone nesnesinin durumunu güncellemeli (kilitli şekilde) ve belki bir **koşul değişkeni** sinyaliyle AI thread'ini durum değişikliğine uyandırmalıdır. Bu mimari dikkatlice tasarlanmalı ve gerekirse kilit mekanizmaları yeniden gözden geçirilmelidir (örn. bir drone bağlantı thread'i ile AI thread'inin aynı anda farklı kilitleri tutup deadlock oluşturmaması için).

11. Bu adım sonunda, örnek bir senaryo olarak tek bir makinede bir sunucu programı ve birden fazla drone istemci programı başlatılarak, istemcilerin sunucuya bağlanıp harita üzerindeki görev koordinasyonunu gerçekleştirdiği doğrulanmalıdır.

12. **Görselleştirme ve İzleme Araçlarının Uyarlanması:** İstemci-sunucu modeline geçişle beraber, görselleştirme yaklaşımını gözden geçirmek gerekir. İki olası strateji vardır:

13. **Sunucu Tabanlı Görselleştirme:** En kolay yöntem, mevcut **SDL görselleştirme** kodunu sunucu uygulamasına entegre etmektir. Sunucu, tüm drone'ların konum ve durumlarını zaten bildiği için halihazırdaki `draw_map()` fonksiyonunu kullanarak kendi penceresinde durumu çizebilir. Bu yaklaşım hızlı bir çözümdür; ancak sunucunun grafik arayüzle çalışması, sunucunun bir arka plan servisi olmasını engelleyebilir.
14. **Ayrı İzleme İstemcisi (Dashboard):** Daha esnek bir yöntem, görselleştirmeyi sunucudan ayırmaktır. README'de de opsiyonel olarak bahsedildiği üzere, bir **web tabanlı dashboard** veya ayrı bir istemci uygulaması kullanılabilir [60]. Örneğin, sunucu her durum değişikliğinde (yeni kazazede eklendi, drone göreve atandı, drone konumu güncellendi vb.) bir **broadcast** mesajı yayınlayabilir. Bu mesajlar ya UDP ile hafifçe atılabilir ya da bir **WebSocket** bağlantısı üzerinden gerçek zamanlı iletilebilir. Ayrı bir dashboard istemcisi de bu verileri alarak harita üzerinde çizim yapabilir. Eğer web tabanlı yapılırsa, harita ve nesneler HTML5 Canvas veya benzeri bir teknoloji ile çizilip, WebSocket ile sunucudan JSON formatında güncellemeler alınabilir.
15. Hangi yöntem seçilirse seçilsin, amaç sistemin durumunu **gerçek zamanlı izleyebilmek** olmalıdır. Geliştirici ekibi için ayrı bir izleme aracı, sistemin kapalı kutu olmaktan çıkmasını ve özellikle dağıtık ortamda (drone'lar farklı makinelerde dahi olsa) merkezi olarak durumun görülebilmesini sağlar. Bu adım, son kullanıcı uygulaması açısından da değerlidir; örneğin bir operasyon merkezinde görev yapan personel, bu arayüzden hangi drone'un nereye gittiğini anlık takip edebilecektir.

16. Bu aşamada, görselleştirme aracının da kendi başına testleri yapılmalı, ağ üzerinden gelen verilerle yerel simülasyondaki görselleştirmenin tutarlılığı doğrulanmalıdır. Gerekirse, performans için çizim optimizasyonları veya fazla veri gönderimini engellemek adına sunucu tarafında **ölçülü güncelleme** (örneğin her küçük hareketi değil, anlamlı değişiklikleri gönder) stratejileri düşünülebilir.

17. **Performans, Ölçeklenebilirlik ve Dayanıklılık İyileştirmeleri:** İstemci-sunucu yapısı çalışır hale geldikten sonra, sistemi daha büyük ölçeklerde ve gerçekçi senaryolarda test etme ve güçlendirme aşamasına geçilmelidir:

18. **Yük Testleri:** README'de belirtildiği gibi, sistemi **50+ drone** ile test etmek önemli bir kriterdir [63]. Bu kapsamda, simülasyon ortamında 50 veya daha fazla drone istemcisi aynı sunucuya bağlanarak sistem gözlemlenmelidir. Bu testler sırasında sunucunun CPU kullanımı, bellek tüketimi ve görev atama gecikmeleri gibi metrikler ölçülmelidir. Eğer sunucu tarafında darboğazlar görülürse (örneğin tek bir AI thread'i bu kadar drone için yetersiz kalabilir), mimaride iyileştirmeler yapılabilir. Örneğin AI görev dağıtımını paralelleştirmek veya kazazede

listelerini bölgesel hale getirerek (her bölge için ayrı bir thread) ölçeklenebilirliği artırmak düşünülebilir.

19. **Hata Toleransı:** Gerçek dünya senaryolarında drone'lar bağlantı kaybedebilir, bataryası bitebilir veya yeni drone'lar sonradan eklenebilir. Sistemin bu durumlarla başa çıkabilmesi için mekanizmalar geliştirilmelidir. Örneğin, sunucu tarafında her drone için bir **timeout** mekanizması konulabilir – belirli bir süre boyunca bir drone'dan **heartbeat** veya güncelleme gelmezse o drone bağlantısı kopmuş sayılıp listelerden çıkarılır ve eğer o drone bir görev yürütüyorsa ilgili kazazede tekrar bekleyenler listesine eklenir (görev yeniden atamaya açık hale gelir). Bu tür logic'ler sunucu içinde uygulanmalıdır. Ayrıca, yeni bir drone sunucuya bağlandığında onu mevcut görevlere dahil etmek mümkün olmalıdır (mevcut mimari zaten yeni bağlantıları listeye eklemeye uygun).
20. **Veri Yapısı İyileştirmeleri:** Yük arttıkça, kullanılan veri yapılarının performansı önem kazanır. Mevcut `List` implementasyonu her ekleme ve çıkarma için lineer arama yapmıyor olsa da, en yakın drone'u bulma işlemi drone sayısına göre lineer zamanda gerçekleşmektedir [17] . 100 drone civarında bu sorun olmasa da, drone sayısı çok artarsa daha optimize bir veri yapısı (örneğin konuma göre bölünmüş listeler veya ağaç yapıları) düşünülebilir. Ancak projenin kapsamı dahilinde bu optimizasyonlar ikincil önceliktedir; öncelik doğru çalışırlık ve sağlamlıktadır.

21. **İstatistik ve Kayıtlar:** Bu aşamada sistemin performans göstergelerini ölçmek için kod içine **loglama** ve sayaçlar eklenebilir. Örneğin her bir kazazedenin bekleme süresi (drone atanana kadar geçen süre) hesaplanıp ortalama/dağılım çıkartılabilir, her bir drone'un ne kadar süre idle kaldığı vs. izlenebilir. README'de not düşülen `log_performance()` gibi bir fonksiyon gerçeklenerek bu veriler toplanabilir [59] . Bu sayede, yapılan iyileştirmelerin etkileri sayısal olarak değerlendirilebilir ve sistemin zayıf noktaları tespit edilebilir.

22. **Ek Özellikler ve Son Rötuşlar:** Son olarak, proje hedeflerine ulaştıktan sonra isteğe bağlı bazı geliştirmeler yapılabilir:

23. **Önceliklendirme ve Akıllı Atama:** Kazazedelere önem derecesi atamak (ör. durumu kritik olanlara öncelik vermek) veya drone'ların görev önceliğini optimize eden daha karmaşık algoritmalar (ör. bazı drone'lar hızlı ancak kapasitesi sınırlı olabilir, bu gibi durumları modellemek) gibi AI geliştirmeleri eklenebilir [64] . Bu, proje kapsamını ileriye taşıyarak akademik araştırma konularına bile dönüşebilir.
24. **Gerçek Ortam Entegrasyonu:** Simülasyon başarılı olduktan sonra, gerçek drone donanımları veya daha karmaşık simülatörler ile entegrasyon düşünebilir. Örneğin, pozisyon güncellemelerini gerçek GPS verisinden alan bir modül veya drone'ları kontrol etmek için gerçek bir uçuş kontrol arayüzü eklemek, projenin prototipten ürüne dönüşmesi yolunda adımlar olabilir.
25. **Dokümantasyon ve Kullanım Kılavuzu:** Projeyi kullanacak veya geliştirecek kişiler için ayrıntılı bir dokümantasyon hazırlanmalıdır. Kurulum adımları, yapılandırılabilir parametreler (harita boyutu, drone sayısı, sunucu IP/port vs.), mesaj formatları ve örnek kullanım senaryoları bu kılavuzda yer almalıdır. README dosyası bu anlamda genişletilip güncellenebilir veya ayrı bir Wiki oluşturulabilir.
26. **Kod Temizliği ve Yapısallaştırma:** Son olarak, proje büyüdükçe kodun modülerliği ve temizliği önem kazanır. Gerekirse kod tabanı bir **CMake** projesine dönüştürülebilir, daha derli toplu bir klasör yapısı (örn. `src/`, `include/` dizinleri) benimsenebilir. Unit test'ler eklenerek kritik fonksiyonların doğruluğu otomatik şekilde test edilebilir.

Bu yol haritasını takip ederek, geliştiriciler proje hedeflerine adım adım ulaşabilirler. Önce thread tabanlı simülasyon tamamen düzgün çalışır hale gelecek, ardından gerçek soket tabanlı istemci-sunucu modeline geçilecek, sonrasında ölçek ve güvenilirlik arttırılacaktır. Böylece **Emergency Drone Coordination System**, eğitim amaçlı bir ödev projesinden, gerçek dünyadaki senaryolara yakın, güçlü

ve kullanılabilir bir sistem haline getirilebilir. Her aşamada alınan sonuçlar değerlendirilerek gerektiğinde geri beslemelerle iyileştirmeler yapmak da başarının anahtarı olacaktır. Projenin nihai çıktısı, bir kontrol merkezi ile birçok drone'un koordinasyon içinde çalıştığı, **gerçek zamanlı, güvenilir ve ölçeklenebilir** bir drone sürü yönetim sistemi olacaktır. [1] [2]

---

[1] [2] [3] [26] [28] [29] [45] [52] [53] [54] [55] [58] [59] [60] [61] [62] [63] [64] github.com
https://github.com/omeraltinova/emergency-drone-coordination/blob/44d1070e364ce198bbd4aaffc6b9dbcd9a1c6728/README.md

[4] [5] [6] [8] [22] [30] [31] [32] [44] [47] [56] github.com
https://github.com/omeraltinova/emergency-drone-coordination/blob/44d1070e364ce198bbd4aaffc6b9dbcd9a1c6728/controller.c

[7] [35] github.com
https://github.com/omeraltinova/emergency-drone-coordination/blob/44d1070e364ce198bbd4aaffc6b9dbcd9a1c6728/map.c

[9] [10] [11] [12] [36] [37] [48] [49] [51] github.com
https://github.com/omeraltinova/emergency-drone-coordination/blob/44d1070e364ce198bbd4aaffc6b9dbcd9a1c6728/drone.c

[13] [14] [15] [38] github.com
https://github.com/omeraltinova/emergency-drone-coordination/blob/44d1070e364ce198bbd4aaffc6b9dbcd9a1c6728/survivor.c

[16] [17] [18] [19] [20] [39] [40] [41] [42] [50] [57] github.com
https://github.com/omeraltinova/emergency-drone-coordination/blob/44d1070e364ce198bbd4aaffc6b9dbcd9a1c6728/ai.c

[21] [23] [24] [25] [27] [43] github.com
https://github.com/omeraltinova/emergency-drone-coordination/blob/44d1070e364ce198bbd4aaffc6b9dbcd9a1c6728/view.c

[33] [34] [46] github.com
https://github.com/omeraltinova/emergency-drone-coordination/blob/44d1070e364ce198bbd4aaffc6b9dbcd9a1c6728/list.c